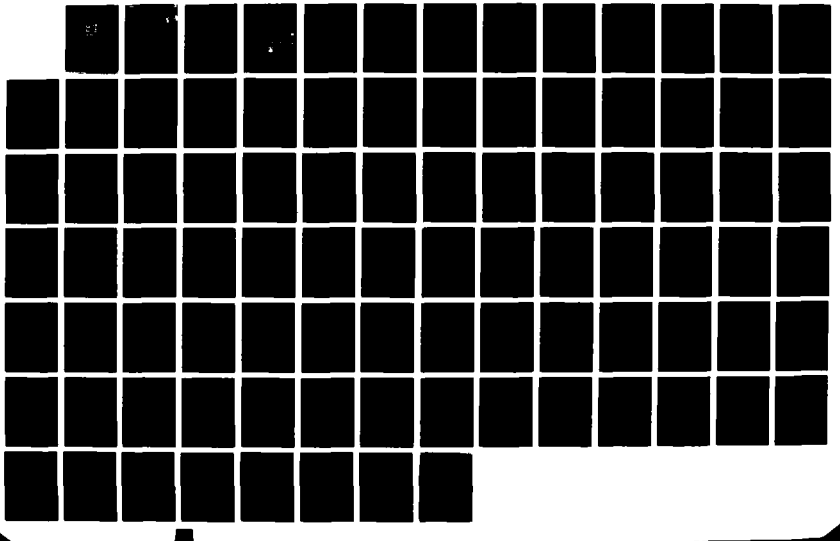
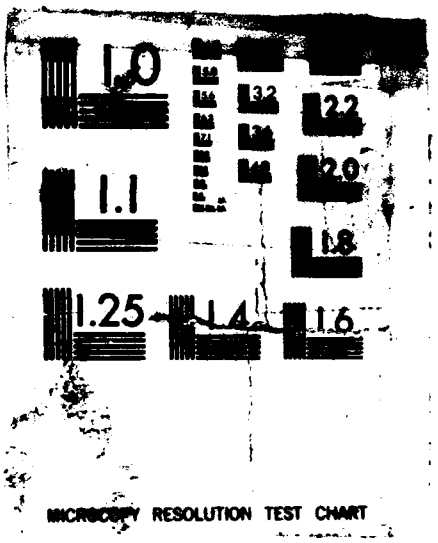
AD-A182 144 REAL TIME PROGRAMS: DESIGN IMPLEMENTATION OF VALIDATION: A SURVEY(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE S LEVI ET AL. APR 87 CS-TR-1837 1/1

UNCLASSIFIED N00014-87-K-0124 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

12

AD-A182 144

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: approved for public relaease; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): CS-TR-1837	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: University of Maryland	6b. OFFICE SYMBOL (If applicable): N/A
6c. ADDRESS (City, State, and ZIP Code): Dept. of Computer Science, University of Maryland, College Park, MD 20742	7a. NAME OF MONITORING ORGANIZATION: Office of Naval Research
	7b. ADDRESS (City, State, and ZIP Code): 800 North Quincy Street, Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-87-K-0124
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

DTIC ELECTE JUN 30 1987 S D E

11. TITLE (Include Security Classification): Real Time Programs: Design Implementation and Validation A Survey

12. PERSONAL AUTHOR(S): Shem-TOV Levi and Ashok K. Agrawala

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	April 1987	85

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The use of real-time systems is widely spread today, and involves very large and sophisticated programs. In addition to the constraints imposed on "regular" very large programs, real-time very large programs are subjected to stringent real-time constraints that the designer tries to meet, to satisfy, and to validate. Those very large programs (systems) are of a very complicated nature, and need special methodologies. This review tries to summarize the methods, approaches, techniques and tools which are used today during a real time system's life cycle. The review deals with three important phases of a real time system: the design phase, the implementation phase, and the validation phase.

Keywords: Computer Program Verification, Naval aircraft, P-7 aircraft

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

CS-TR-1837 April 1987

REAL TIME PROGRAMS: DESIGN IMPLEMENTATION AND VALIDATION

A Survey *

Shem-Tov Levi and Ashok K. Agrawala

Department of Computer Science
University of Maryland
Computer Systems and Analysis Group
College Park MD 20742

(*) This research is supported in part by a contract from The Office of Naval Research to The Department of Computer Science, University of Maryland.

Contract No. N00014-87-K-0241

CS-TR-1837 April 1987

REAL TIME PROGRAMS: DESIGN IMPLEMENTATION AND VALIDATION A Survey *

Shem-Tov Levi and Ashok K. Agrawala

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

Abstract

The use of real-time systems is widely spread today, and involves very large and sophisticated programs. In addition to the constraints imposed on "regular" very large programs, real-time very large programs are subjected to stringent real-time constraints that the designer tries to meet, to satisfy, and to validate. Those very large programs (systems) are of a very complicated nature, and need special methodologies. This review tries to summarise the methods, approaches, techniques and tools which are used today during a real time system's life cycle. The review deals with three important phases of a real time system: the design phase, the implementation phase, and the validation phase.

Contents

Chapter 1

Introduction

The commercialization and cost reduction of microcomputers has resulted in an increased use of real-time systems in a wide variety of fields. Traditional hardware equipment has been replaced by computerized systems, which have provided a more flexible and expandable environment. Military, industrial and medical applications implement most of their required control functions using computerized real-time systems. Examples of such applications are nuclear power plant control, industrial plant control, medical monitoring, digital fly-by-wire avionics and weapon delivery systems. As the use of such systems has spread, the timing requirements have become more stringent, and the reliability requirements have become more difficult to achieve.

Designing real-time systems is thought to be one of the most complex programming activities. When building a model of ascending complexity in designing computer systems [34], one starts with sequential programming, increasing the complexity to the level of concurrent programming, and finally adding the timing notion to reach the level of real-time programming. Combining the reliability requirements and the complexity involved, results in a severely restricted environment. Furthermore, not as in the sequential and concurrent programming, real-time systems are implementation dependent. A program that has been executed successfully when implemented in one environment, does not necessarily satisfy the constraint of another environment. Changes in the computer hardware, communication network, operating system or peripheral device response time, may change the system behavior to a point where it does not satisfy a particular program's requirements any more.

This review tries to summarize the methods, approaches, techniques and tools which are used today during a real-time system's life cycle. The review deals with three important phases of a real-time system: the design phase, the implementation phase, and the validation phase. (Another important phase, the maintenance, is not included in this review). A chapter is dedicated to each of the above phases.

The goal of this review is not to criticize the different approaches which are summarized here, rather it is to highlight interesting aspects which concern the life-cycle of a real-time system. A strong emphasis is layed on new approaches. Some of the approaches are described in more details than others. Methods which are very widely used nowaday, are sometimes briefly mentioned, while new (and sometimes even immature) methods are reviewed in depth. The reason is certainly not the importance of the more detailed ones, but the limitations of this review due to its nature and goals.

Chapter 2

Design Methods for Real Time Programs

2.1 General

Many approaches and methods are used in designing software. This chapter tries to capture past and present works which are used in designing real-time software. Some commonly used methods are not reviewed here. Jackson and Warnier design methods are data structure oriented methods, primarily suited for program level design. Both can neither handle the decomposition into modules or tasks, nor are they appropriate for real-time systems. Higher Order Software methods use functional decomposition, but fail to address tasking and synchronisation issues. Thus, two major methods are examined: a structured design method, and an event-based design method. The methods are divided further according to the modeling they enforce, and to the system statement and analysis approaches they adopt.

The next section of this chapter deals with requirement specification techniques and approaches. The third section describes structured design methods, mainly process based methods, and some problematic issues concerning the application of these methods. The fourth section describes issues that arise when applying a graph-based design method. The fifth section compares the above conventional methods versus the operational method which is more implementation oriented. The last section of this chapter reviews FSA (Finite State Automata) modeling of real-time systems, mainly the Petri Nets Theory and some augmentations applied to it.

2.2 Requirement Specification Methods

A lot of software engineering research has addressed the problem of defining complete and consistent requirements specification methodologies. In this section, a review of two methodologies is given, along with an example of how to implement a third methodology. The principles of these methodologies are important to real-time software design, as they are to other fields. Although some methods emphasize only the documentation part of this early phase of design, other methods provide good tools for structure construction and feasibility checks.

2.2.1 PSL / PSA

The PSL (Problem Statement Language) and PSA (Problem Statement Analyzer) are computer aided tools [32] especially designed for requirement specification statement and documentation of information processing systems. This tool emphasizes the front end of the system for specifying the requirements, and produces a large variety of documents describing the database for the system specifications entered so far. The database is built by steps, and hierarchy can be imposed. The basic structure in the system description in the OBJECT, to which PROPERTIES can be attached with PROPERTY VALUES. The objects are connected by RELATIONSHIPS, while both objects and relationships may be classified with TYPES. PSL produces upon demand eight types of reports for system description:

1. System I/O Flow - The interaction between the target system and its environment.
2. System Structure - Objects hierarchies.
3. Data Structure - Internal relationships between data objects.
4. Data Derivation - The relationships between processes and data objects.
5. System Size and Volume.
6. System Dynamics - System "behavior" within time.
7. System Properties - Distinguishing remarks.
8. Project Management - Project schedules etc.

In addition PSA Produces the following report types:

1. Record of modifications applied to the database.
2. Reference reports - Names, properties, dictionary.
3. Summary reports - Structure and flow.

4. **Analysis reports - Contents comparison, data processing interactions, processing chain.**

For a design of a real-time system, this computerised tool can be used for documentation, but no benefits can be gained for the continuation of the design phase, and for the verifiability of the design.

2.2.2 SREM

SREM is a requirement engineering methodology [4], which was developed for the Balistic Missile Defense Advanced Technology Center by TRW. The methodology adopts the functional hierarchy requirements of MIL-STD-490, in which each process is divided to functions, each of these further divided to subfunctions, and so on. The hierarchy imposes validation difficulties, especially when trying to exercise sub-sub-function through input sequences.

Seven Key Concepts of SREM

1. **A testable requirement must be specified in terms of data input and output. The reason for this concept is that a real-time software is tested by inputting a MESSAGE and extracting the results of its processing (an output MESSAGE) and the content of memory.**

2. **Processing PATHs are sequences which do not contain loops, defined in terms of input MESSAGEs, output MESSAGEs, processing steps, and data utilised and produced.**

3. **VALIDATION POINTS are places where a test may be performed in terms of variables measured on the PATH. Defining the testable variables on validation points assures testability and unambiguity of the requirements.**

4. **R-NET is an integration of all the PATHs that process a given type of stimulus, into a Requirement NETwork. This is a graph model of the computation and the flow. Five types of nodes are used in the graph: processing step node (ALPHA), input/output interface node, and/or flow node, selector node, and validation point node. The unidirectional arcs represent the data flow between the nodes.**

5. **RSL - a formal language for specification of requirements.**

6. **REVS - an automated tool which speeds up and validates the requirements' completeness and consistency.**

7. **Methodology STEPS - which produce intermediate products obeying evaluation criteria for each step.**

Methodology Steps and Their Products

Step 1. Translation.

1. Issues addressed:
   - Adequacy of subsystem performance requirements (DPSPR) for generating processing requirements.
   - Early baselining of the functional requirements.
   - Budgeting and scheduling the activities.
2. Activities in this step:
   - RSL originating requirements entered into REVS database.
   - Generation of R-NETs, DATA, ALPHAs, with traceability back to DPSPRs.
   - Analysis of consistency and completeness of the requirements.
   - Generation of DPSPR problems report.
   - Budgeting and scheduling further activities.
3. Products:
   - DPSPR problem report.
   - R-NETs.
   - ALPHAs.
   - DATA.
   - Functional traceability.
   - Plans.

Step 2. Decomposition.

1. Issues addressed:
   - Preliminary definition of the performance requirements.
   - The incorporation of the processing to satisfy the subsystems constraints into the processing requirements.
2. Activities in this step:
   - Identification of the form of the performance requirements.
   - Specification of the data collected at validation points (software variables).
   - Recording the decisions made, to relate accuracy and timing requirements back to DPSPRs.
3. Products:
   - Refined RSL.
   - Performance traceability.
   - Validation points.

Step 3. Allocation.

1. Issues addressed:
   - Determining the sensitivity of PATH performance to DPSPRs.
   - Establishing the tradeoffs between accuracy and timing of the different PATHs, then selecting an allocation which is not overly restricted.
2. Activities in this step:
   - After establishing the requirements of each of the PATHs, the requirement and its test are written in RSL.
   - In complicated systems a functional simulator is developed.
3. Products:
   - Performance sensitivity.
   - Performance statement.
   - Process Performance Requirements (PPR).
   - Functional simulation.

Step 4. Analytical Feasibility Demo.

1. Issues and Activities:
   - Example algorithms are implemented to demonstrate that critical processing requirements can be satisfied. This is done before attempting the design of an algorithm for the real-time software.
   - A direct check is provided by the above, the algorithms that satisfy the PPR, in fact meet the originating requirements - the DPSPR.
2. Products:
   - Example algorithms.
   - Simulator.

Conclusion

The SREM methodology allows specifying complete and consistent requirements to both the system and its subsystems. The R-NET graph model resembles more advanced models of FSA, and allows better visibility of relations and testability properties. Yet, most of the performance decisions and the structure built-up are manual, and depends highly on the designer's skills. The use of a functional simulator to verify performances is misleading due to common mode errors (see section 4.4.2), and to the high cost of providing it on time.

2.2.3 Example: A-7 Requirements Specification

Another approach for requirements specification is introduced in the redesign of the avionics software of the A-7 Navy's aircraft. The method introduced [18] is used by the Naval Research Lab and the Naval Weapon Center. The flight program which was documented is a part of the Navigation and Weapon Delivery System of the aircraft. The program has high accuracy requirements and stringent real-time constraints. It receives input data from the aircraft sensors and from operational switch panels, and controls many devices (e.g. the Inertial Measurement System, the Head-Up display, the Doppler, Barometer etc.). The program's main tasks are to calculate the navigation information and to control the weapon delivery.

Requirements Document Objectives

The objectives of a document that integrates the requirements of a software system, as interpreted by Parnas et al, are listed below.

1. Specify external behavior only, without implying a particular implementation.
2. Specify constraints on the implementation, especially the details of hardware interfaces (usually the case with computer embedded systems).
3. Installing changes should be easy.
4. Ability to serve as a reference tool.
5. Record forethought about the life cycle of the system.
6. Characterize acceptable response to undesired events.

Requirements Document Design Principles

1. State questions before trying to answer them.
2. Separate concerns: organise the document such that any project member could concentrate on a well-defined set of questions.
3. Formality should be used as much as possible, in order to obtain precision, consistency, and completeness.

Techniques for Describing Hardware Interfaces

- *Organisation of Data Items:* A data item is a unit concerning an input or an output that changes value independently of other inputs or outputs.

- *Symbolic Names* (for Data Items and Values)*:* Data items contain two kinds of information: arbitrary details and essential characteristics. Essential information must be expressed in a way that would allow using it from the rest of the document, without referencing the arbitrary details.

- *Templates for Value Descriptions:* Describing each data item in ad hoc fashion produces inconsistencies between documents. The existence of templates provides the following features: values description is easier, there is a consistency between documents, and standards of completeness may be applied uniformly to all items of the same type.

- *Input Data Items Described as Resources:* The input data items are described as if using an inventory of available resources to solve the problem. This description is independent of software use.

- *Output Data Items Described as Effects:* Most of the output data items are described as effects on external hardware.

Techniques for Describing Software Functions

- *Organising by Functions:* Functional hierarchy is adopted as for MIL-STD-490. Two classes of software functions are distinguished: periodic functions and sporadic (on-demand) functions. The distinction is useful in cases where different performance and timing constraints are imposed for each.

- *Output Values as Functions of Conditions and Events:* A condition is a predicate that characterizes some aspect of the system for a measurable period of time. An event occurs when a condition changes its truth-value. Hence, events are associated with instants of time, whereas conditions are with time intervals. Events designate start and stop of periodic functions, and they trigger sporadic functions.

- *Consistent Notation for Operating Conditions:* Maintaining a consistent notation has crucial importance in requirements document. The A-7 example provides a notational standard in which conditions, events and text macros are well defined and distinguished.

- *Using Modes to Organize and Simplify:* Modes (i.e. classes of system states) help in simplifying and organising the hierarchical structure, by introducing a higher abstraction level. Furthermore, a transition list which includes entries from one mode to another, allows detection of illegal transitions as well as serving as a control tool.

- *Special Tables for Precision and Completeness:* The A-7 document uses two special tables to express information precisely and completely. A condition table is used to define an output value upon a specified active

mode and a condition that occurs within that mode. An event table shows when a sporadic function should be performed, or when a periodic function should be started or stopped, with respect to the currently active mode.

List of Undesired Events

A classification of undesired events, as given by Parnas, is sketched below. As mentioned before, an acceptable response to any of these occurances should be specified, and not left to the programmer to invent.

1. Resource failure.
   - Temporary.
   - Permanent.
2. Incorrect input data.
   - Detected by examining input only.
   - Detected by comparison with internal data.
   - Detected by user realising he made a mistake.
   - Detected by user from incorrect output.
3. Incorrect internal data.
   - Detected by internal inconsistency.
   - Detected by comparison with input data.
   - Detected by user from incorrect output.

2.3 Process Based Structured Design Methods

2.3.1 Theoretical Description

A theoretical examination of the process based methods is given in [26].

Types of Process Based Methodologies

Mok distinguishes two types of design methodologies in applying the traditional process based design method:

1. The *virtual processor* methodology: Each process is assumed to have a dedicated processor (see also [34]). The objective of this approach is to have a proven bounded execution time, while assuring properties of no-deadlock, fairness, etc. Timing constraints are therefore left to be solved by the scheduling strategy. This solution does not address issues such as communication bottle-neck, which then has to be addressed through

hierarchy of processors that often reduces the problem, but does not always solve it.

2. *Processor sharing* methodology: Processors are assumed to share resources. The sharing is subjected to known scheduling disciplines and usage restrictions. A possible usage of priority discipline is expected in the scheduling (e.g. Earliest Deadline Algorithm). A difficulty that arises is that when processes may request service with no a priori knowledge of request times, then not all feasible sets of timing constraints can be satisfied by any one multi-processor scheduling algorithm.

Description of the Theoretical Model

Some problems are highly relevant to hard real-time environment:

- Decomposition of the computational requirements.
- Dictation of the processing scheduling discipline.
- Adequacy of the concurrency control mechanism.

In order to analyze the system behavior Mok introduces the following model:

- Let M_p be the set of periodic processes in the system.
- Let M_s be the set of sporadic processes in the system.
- $M = M_p \cup M_s$.
- Each process T_i is a triple (c_i, p_i, d_i), where: c_i denotes the computation time of the process, p_i denotes the period, and d_i denotes the deadline.
- $\forall i : c_i < d_i < p_i$.

Although a sporadic process has no specific period, its p_i stands for the maximal frequency in which it is allowed to appear.

2.3.2 Scheduling Using a Process Based Model

Scheduling Requirements

A necessary condition for scheduling is that the sum of all utility factors of the processes is less than the number of available processors. In other words:

$$\sum_i (c_i/p_i) < \#\text{ of avail processors.}$$

The scheduling in real-time environment requires that all the constraint of the periodic and the sporadic processes are continuously satisfied. The most general scheduling construct involves two schedulers: an off-line scheduler which

"dictates policy", and a run-time scheduler which is completely on line and obeys the policy dictated by the off-line. For theoretical analysis purposes, Mok defines a "clairvoyant" scheduler: it has special insight and can predict the unpredictable.

Single Processor Scheduling

The most obvious scheduling algorithm is the "earliest deadline" algorithm. The scheduler chooses to execute the process whose deadline is the earliest to happen. A better approach is to use a scheduling policy which chooses the process whose maximal delaying possibility is the lowest. This approach is called the "least slack" algorithm, where the slack of a process at time t is defined as the maximum time which a run-time scheduler can delay it, without disobeying the constraints.

$$slack(P_i, t) = max(d(t) - t - c(t), 0)$$

Mok proves the following theorems concerning a single processor scheduler:

- The least slack algorithm can be used as a totally on-line optimal run-time scheduler, under the assumption that the scheduler can choose to preempt a process, by any other ready process, at any integral time instance.
- Where there are mutual exclusion constraints, it is impossible to find a totally on-line optimal run-time scheduler.
- Let $M = M_p \cup M_s$ be an instance of a processes model, and let $q_i = d_i - c_i$ be the nominal slack of each process. Replace every $T_i = (c_i, p_i, d_i) \in M_s$ by a periodic $T_i' = (c_i, p_i', d_i')$, such that $p_i' = min(p_i, q_i + 1)$ and $d_i' = c_i$. If M' (which is created by the replacement) can be successfully scheduled, then M can be scheduled without a priori knowledge of the request times of M_s.

Multi-Processor

In a multi-processor environment, (probably using a rendezvous mechanism to synchronise between communicants) the earliest deadline algorithm may fail. Mok suggests it can be fixed by revising the deadlines dynamically, as follows:

1. Sort scheduling blocks generated in [0,L] in reverse topological order.
2. Initialise the deadline of the k'th instance of $T_{i,j}$ to $(k-1)p_i + d_i$.
3. Revise the deadlines in reverse topological order by

$$d_k = Min(d_k, (d_k' - c_k' : k- > k')),$$

where k and k' are scheduling blocks.

This modification allows the following theorem:

- If a feasible schedule exists for an instance of model, restricted by rendezvous constraints, then it can be scheduled by the earliest deadline algorithm, modified to schedule the ready process which isn't blocked by a rendezvous, and has the nearest dynamic deadline.

2.3.3 Requirement of a Structured Design Method

A structured design method decomposes its modules hierarchically. When applied to a real-time system, the method is required to provide the following ([16]):

1. *Data-flow-oriented design.* A structured design consists of two main components: (1) Two sets of criteria, *cohesion* and *coupling*. (2) *Top down* decomposition of a system into modules. The objective of the design is to produce a system in which modules have high cohesion and low coupling. In order to examine these properties, data-flow approach is appropriate [22], showing the functional modules (*transforms*), the data flow between them, and the data stores accessed by them. Furthermore, real-time systems are usually data flow oriented. Two design approaches may be distinguished:
   - *Transform Centered Design*, in which the major streams of data are identified as they flow, transformed from external input to external output.
   - *Transaction Centered Design*, applicable where the data flow consists mainly of control-information, i.e., data which is passed to a transform initiates an action (or a sequence) based on the incoming data.
2. *Task Synchronization.* Two kinds are mostly used:
   - Mutual exclusion : shared data can be concurrently accessed by two or more tasks, while the access is controlled by means of semaphores.
   - Cross simulation : one task is awaiting a signal from another in order to proceed.
3. *Task Communication.* Message communication is the most commonly used method. Task communication can be closely coupled (a response is expected in order to continue), or loosely coupled (with the use of message queues). A message consumer that finds an empty queue, waits until a message arrives. There are three ways to implement the communication:
   - Using the operating system primitives.
   - Using a language facilities [2].

- Using a module which provides the services (itself using operating system primitives, as implemented in MASCOT channels).

4. *Information Hiding Concept.* This concept (by Parnas) is powerful in leading to a highly modular structure. The idea is that each key is known only to one module, hence the shared data is kept to minimum. Modules are therefore more self contained (thus more modifiable and more maintainable). The cost of this approach is in the overhead in accessing a data structure via a function rather than directly.

5. *State Dependency.* Most approaches provide dependency of taking an action on input data. Some (especially found in transaction centered approach implementations) fail to allow dependency of taking an action on the system's state. A structured design method should incorporate state dependency as well as data dependency.

2.3.4 DARTS

DARTS is a Design Approach for Real-time Systems, proposed in [16] as an extension of the older structured design methods, to include task structuring as well as task interface definitions. DARTS was developed by General Electric, and was applied to two projects: a robot controller and a vision system. When the paper was written the robot was already in the integration phase, and the application of the method was considered successful. DARTS starts with the requirements specification, and its first phase is analysing the data flow.

Data Flow Analysis

Analysis of the data flow through the system yields the determination of the major functions that are needed. Data flow diagrams are produced, and decomposed to identify the major subsystems and the major components of each subsystem. The data flow graph produced consists of: transforms (represented by bubbles), data depositories (represented as data stores), and data flow between transforms (represented by directed arcs) [1].

Tasks Decomposition

Concurrency properties of the previously identified transform can be derived from the data flow graph, using the asynchronous nature of the transforms within the system as the main consideration of the decomposition process. The criteria used in DARTS for deciding whether a transform should be a separate task, or grouped with other transforms into one task, are given below.

[1] Although a graph description and analysis is performed, the method as a whole is completely process/task oriented.

1. Dependency on I/O: If the speed is dictated by a slow I/O device, then a separate task.
2. Time Critical Functions: If a high priority is distinguished for a particular function, then a separate task.
3. Computational Requirements: If a function is identified to have intensive computations, then a separate task – to allow a work mode of spare cycles "stealing".
4. Functional Cohesion: If functions are found to be closely related, then grouped into one task, to reduce system overhead. Within a task, modules can be distinguished for functional cohesion at the module level.
5. Temporal Cohesion: If functions are found to be triggered by the same stimulus, then grouped into one task, distinguished as different modules within the task.
6. Periodic Execution: If a transform needs to be executed periodically, then a separate task, activated at regular time intervals.

Interfacing Tasks

TCM - Task Communication Module. A typical TCM contains a data structure and an access procedure to that structure. The access procedure controls also the mutual exclusion and the synchronization features, which may use operating system primitives. The TCM always runs on the task that involves it. Two different types of TCM are provided by DARTS:

1. MCM - Message Communication Module: Supports both closely coupled and loosely coupled communication. In the case of loosely coupled communication, the message queue includes binary semaphores for mutual exclusion. In closely coupled communication, the queue size is forced to one.
2. IHM - Information Hiding Module: Used mainly in cases of shared data. The IHM defines both the data structure it hides and the access procedure to it.

TSM - Task Synchronization Module. Typically, a TSM has a nature of a supervisory module of a task (could be the "Main" module which has the same nature). The TSM uses synchronization events when no actual information has to be exchanged. In DARTS the primitives for signaling an event, and for awaiting an event, are provided by the operating system.

Interfacing Summary. In DARTS the tasks interfaces are implemented as follows:

1. A data flow between tasks is interpreted as one of the following:
   - A loosely coupled messages queue, handled by an MCM.
   - A closely coupled message/reply, handled by an MCM.
   - An event signal, if only occurance notification is required.
2. A shared data store is handled by an IHM.
3. A task that waits for an event, may need a TSM.

Task Design

Structured Design. Each individual task represents a sequential program. The design of the task starts with the flow analysis at the task level, followed by applying one of the two possible structured design approaches stated above; According to the nature of the task, either a transform centered design approach or a transaction centered design approach is applied.

State dependency. DARTS uses a State Transition Manager (STM), designed as a TCM of the IHM type. The state transition table is maintained by this module, hidden from the calling task. While responding to a transition request, the request validity is checked by the access procedure, and when confirmed the module executes the transition. In order to ensure the atomicity of the transition execution, as well as to achieve fast transitions, an approach is recommended by the author in [16]. The idea is to increase the task's priority when STM is entered, and restore the old priority when the STM is exited.

2.3.5 Task Allocation Scheme for a Real-time System

As mentioned above, many design methods fail to address the task construction, and thus are not suitable for designing real-time systems. That is the reason for including the following scheme [25] in this review. The task allocation scheme was developed by TRW, and was applied successfully in the BMD (Ballistic Missile Defense) project.

Port-to-Port Execution Time

Software tasks in time-critical real-time systems are usually divided into several threads, and each of the threads must satisfy an execution-time constraint, denoted as the port-to-port processing time. In the above application, 23 tasks were divided into 7 threads. Execution time of a thread consists of four components:

1. Execution time of the task on the processor: which depends on the task size and the processor MIPs rate.

$E_i = size(task_i)/MIPs$ rate of the processor.

2. The network and operating system overhead (NO), which is used for concurrency control, integrity checking, recovery check-point update, etc.

3. Inter-processor communication (IPC), which is higher if communicants reside on different processors.

4. Waiting time (WT) which is consumed when the task waits in the processor enablement queue. This figure depends highly on the sizes and number of tasks, the processor load, and the number of enablements. (Especially if large tasks are assigned to the same processor.)

Therefore,

$$E_T = \sum_i (E_i) + NO + IPC + WT.$$

For a given network, NO and the number of enablements is relatively a constant. Hence, in order to reduce E_T the following steps should be adopted:

- Reduce WT: large tasks should be assigned to different processors.
- Reduce IPC: tasks with high IPC cost with each other should be assigned to the same processor.
- Reduce E_i: large tasks should be assigned to processors with higher $MIPs$ rate.

Allocation Model

The above considerations yield the following sequence of activities in designing an allocation scheme:

- A set of constraints is determined, to reduce the waiting time and the task execution time.
- A cost function that measures the IPC cost is formulated.
- An algorithm that searches for the allocation with the minimum total cost is determined.

The above activities are performed in the following method:

1. Information is entered to the model about the tasks (sizes, execution frequency of each task, number of data units exchanged between each pair of tasks) and the network (Inter-processor distance, constraints).

2. Constraints are imposed on the model:

- Task preference matrix: Certain tasks (out of m) can be executed only on specific processors (out of n). These restrictions are formulated as an $m \times n$ matrix of 0's and 1's. $X_{i,j} = 0$ means task i can't be assigned to processor j. $X_{i,j} = 1$ means no restriction on task i with respect to processor j.
- Task exclusive matrix: Defines mutually exclusive tasks, and expressed as an $m \times m$ matrix. $X_{i,j} = 0$ means no constraint between task i and task j. $X_{i,j} = 1$ means task i and task j can not be assigned to the same processor.

3. The cost function is formulated. It depends on the following parameters:
   - Task coupling factors $c_{j,k}$: number of data units transfered from task j to task k.
   - inter-processor distance $d_{q,p}$: the cost of a transfer of one data unit from q to p.
   - tasks quadratic assignment formulation: $(X_{j,p} = 1)$ means task j assigned to processor p.

2.4 Graph Based Structured Design Method

A graph based design method uses techniques taken from graph theory in order to analyse and construct the system. The main difference between this method to the structured method is that all data dependencies are explicitly expressed. This method is superior to the structured method in its capability to identify operations on data which are common to many timing constraints.

2.4.1 Graph Based Theoretical Model

Mok (in [26]) provides a theoretical examination of the graph based design approach, and the problems which arise when applying a scheduling algorithm based on this model.

Model Description

A graph based model M is an ordered pair (G, T), where G is the communication graph of the system, and T is a set of timing constraints that satisfies $T = \{T_p\} \cup \{T_a\}$. The subsets T_p and T_a are the periodic and asynchronous (sporadic) constraints respectively. The communication graph $G = (V, E, Wv)$ is a digraph which may contain cycles, in which V is a set of vertices, E is a set of edges (directed arcs), and Wv denotes a function that assigns a non-negative weight to each node in V. Each timing constraint in T is represented by a triple (C, p, d), where C is a timing constraint acyclic graph (compatible with

G), and p, d represent the periodic and deadline constraints respectively. In the case of asynchronous (sporadic) constraint, p represents the maximal occurance frequency allowed.

The timing constraints graph expresses the precedence relations between computational events that must be kept in order to satisfy the timing constraints. Execution of a functional element is denoted by a node, and data transmitted in the communication graph (G) by a directed arc. The computation time of a timing constraint (C, p, d) is obtained by summing the weights of all the nodes in C. If (C, p, d) is activated at time t, then C must be executed at $(t, t + d)$.

C is said to be executed in a time interval L, if a subset S of the (multi)set of the functional elements of C was executed in L and forms a partial order such that:

1. There exists a bijective mapping between the functional elements in S and C.
2. Under this mapping the partial order of S is consistent with the acyclic graph C.
3. If the functional elements are distributed, and there exists an edge $u \rightarrow v$, then an execution of C must include a transmission of the latest output of u to v, before v is executed in L.

Pipelined Order Requirement

A pipelined order is interpreted by Mok as:

1. If two executions of a functional element have two distinct start-times, then the one with the earliest start-time must also finish first.
2. No message-overtaking (desequencing) is allowed.

Defining c as the execution time of C, allows mapping $T = (C, p, d)$ to $T' = (c, p, d)$. Then, creating a monitor for each functional element of C that occurs in more than one timing constraint, allows imposing the pipelined order requirement. Decomposition of each functional element into subelements, whose sum of execution times is approximately the same as those of the functional element, is as a matter of fact software pipelining. This pipelining improves the efficiency by taking advantage of operations that are common to many timing constraints.

Latency and Static Scheduling

Mok defines some metrics that are needed for defining and analysing a static scheduler. First, an *execution trace* of a processor is defined as a mapping from the non-negative numbers to the set of all the functional elements in G and the idle. We denote this mapping as F. Recall that the graph based

model $M = (G, T)$. For example: $F(i) = u$ if the functional element u in G is executed in the time interval $(i, i+1)$, and $F(i)$ =idle if the processor idles in $(i, i+1)$. Second, an execution trace is said to have *latency* k with respect to a timing constraint, if the execution trace contains an execution of the timing constraint in any time interval of length greater or equal to k time units. A *static schedule* is defined as a finite list Y of symbols from the set of vertices of G and the idle. The latency of a static schedule is defined with respect to a round-robin generated schedule. Y has a latency of k time units with respect to the constraint (C, p, d) IFF the execution trace generated by a round-robin scheduler, repeating Y ad infinitum, has a latency of k. In order that a static scheduler would be feasible with respect to a set of asynchronous constraints T_a, it should have a latency d with respect to every (C, p, d) in T_a. Mok proves the following properties for a graph based model (G, T):

- The existence of a feasible static schedule with respect to T in the cases where a latency d exists for every (C, p, d) in T.
- Proving the existence of a feasible static schedules when the above requirement is not satisfied, is NP hard even for relatively simple cases. Only application of additional constraints on relations between computation times and deadlines allows proving the existence of a feasible static schedule.

2.4.2 Data and Control Flow Graphs

A graph modeling technique which is commonly used is the Data Flow Graph [22], and its description and usage possibilities are give below.A bi-digraph is used to model the system. A *control flow graph* describes the structural behavior of the program and the control flow during execution, while a *data flow graph* (corresponding to each execution sequence) describes the data behavior during this execution.

Data Flow Graph Model Description

In the control graph: the *vertices* represent *control points*, and the directed *arcs* represent *actions* or control transitions. In the data flow graph: there are two types of *vertices* - *data items* and *operations*. The vertices are connected by directed *arcs* describing the data *flow*. A data flow graph corresponds to an execution sequence $S = (a_1, .., a_n)$ in the control flow graph, by attaching to each arc a_i a mapping of input variables $(X_1, .., X_k)$ into output variables $(Y_1, .., Y_m)$. This functional relation is the vertex of type "operation" which appears in the graph. The graph, which may be very large in case of a complex program, may be reduced by means of abstraction levels, merging data items to vectors, and sequential actions into control segments.

Usage of Data Flow Graph

This graph may be used to both demonstrate structural properties and verify some performance ones.

- Independent data items may be detected, and point to the distributed implementation that requires less communication traffic. The problem becomes a partitioning problem which requires that the number of arc-cuts is minimised.
- Each execution sequence with its corresponding data flow graph encounters all the information needed for numerical error-bound analysis.

2.5 Operational and Event Based Design Methods

An event based design method describes the system as a mechanism which responds to events fed to it from the external environment. It was already mentioned in this review that a real time system has high dependency on the implementation of the design. The main difference between the event based approaches is to what extent the design is separated from the implementation.

2.5.1 Event Based Model

Event based model of a system [9], separates the systems's properties into two major categories: *behavior* which mainly concerns the external view of the system, and *structure* which reflects the internal view of the system. Proving the correctness of a system is translated to a consistancy check between the behavior and the structure. Orthogonality between properties allows verification of each independently, and thereby avoids "exponential state explosion" when coming to derive test sets.

Description

The model is constructed from events and their relations:

1. An *event* is an *instantaneous* (takes zero time) *atomic* (happens completely or not at all) state transition in the system's computation history.
2. *Time ordering* of events is achieved with the *precede* ($\longrightarrow$) relation, which is a partial ordering found also in [24]. Event $e1$ precedes event $e2$($e1 \longrightarrow e2$) if:
   - $e1, e2$ are events at the same process (an autonomous computation node, having its own local clock) AND $e1$ comes before $e2$, OR

- $e1, e2$ are events at different processes, AND $e1$ is a *send message* event, AND $e2$ is a *receive* event of the very same message.

This partial ordering ensures that $e1 \longrightarrow e2$ implies that $e1$ happens before $e2$ by any measure of time. (It is not IFF!). The ordering is transitive, irreflexive and anti-symmetric.

3. *Causality ordering* is achieved with the *enable* ($\Longrightarrow$) relation, which is also a partial ordering, and is defined as follows:

   - Event $e1$ enables event $e2 (e1 \Longrightarrow e2)$ IFF the *existence* of event $e1$ will *cause occurance* of event $e2$ in the future.

   This ordering is also transitive, irreflexive and anti-symmetric.

4. *Domains* are used to ease the specification procedures. Three of them are used to classify the events: the system, the environment, and interface ports. The main idea is to construct the specifications in a scheme, where the system and its environment interact with each other using message communications that are exchanged via uni-directional interface ports. In-ports are used to enter messages from the environment to the system, and out-ports are used for the opposite direction. In every port a total ordering of events is imposed by assigning a distinct ordinal number to each interface event.

2.5.2 Design with Event Based Approach

The system specification is entered as a set of axioms, applying top down approach. The behavior specification is done using EBS language, which is based on the event concept and first order logic. The behavior (external view) is specified first, and then it is decomposed into a design structure (i.e. the internal view). Finally, the model is verified with a consistency proof, to assure that the structure satisfies the behavior requirements. The decomposition process uses three construct types to describe the design structure: the sub-system, the link, and the interface-definition. The objective is to decompose the design structure such that sub-systems communicate with each other through uni-directional links, and the communication with the environment is done through the interface definitions. The constructs are specified as follows:

A sub-system: A set of events, a subset of the system events set. The computation is defined by the behavior specification.

A link: A connection between an out-port of the subsystem to an in-port of another subsystem; as in "connect(X,Y)==Z", where Z is a link connecting out-port X to in-port Y.

An interface-def: A definition of a sub-system's port as a system's port; as in "X==Y" where a sub-system's port X is the system's port Y. Both ports should have the same direction.

An example of a design structure can be:

Def System $S(I : in - port; O : out - port)$;
  Structure
    Sub system $SS(II : in - port; OO : out - port)$;
      Behavior
      ...
      end Behavior;
    Sub-system ...
      ...
    Network
      $connect(X, Y) == Z$;
    end Network;
    Interface
      II==I;
    end Interface;
  end Structure;
end System;

2.5.3 Operational Approach

The PAISLey project uses a new approach in real time systems design, in which the idea of an implementation dependency is expressed in a strategy called *operational approach* [35].

Principle

The main idea is that external behavior and internal structure may interleave. In order to maintain generality, the operational approach separates the problem-oriented structure of the operational specifications from pure implementation considerations. The operational specifications themselves are written in an operational specification language, which is *executable* and prevents ambiguities. The executability feature of the specification provides a tool for early results examination, and can be regarded as a functional simulator that corresponds to the specifications constraints. Automated translation (from specification statement to an implementation code) is highly feasible, since problem-oriented internal constraints are taken into account. This transformational implementation would therefore guarantee correctness of the produced code. P. Zave compares the operational approach with the conventional approach in [35]. A summary of this comparison is given below.

Operational Versus Conventional Design Approach

- **Advantages of the conventional approach:**
  1. **The *informal* specification, written in English or another natural language, can be understood by everyone.**
  2. **Organisational benefits (like *milestones*) are easily obtained.**
- **Advantages of the operational approach:**
  1. **The specifications are formal and non-ambiguous.**
  2. **A rapid prototyping is available automatically.**
  3. **Verification is easy due to fixed transformational implementation.**
  4. **The realisation phase is highly automatable.**
  5. **The conflict between efficiency aspects and maintenance aspects is a major concern.**
- **Weaknesses of the conventional approach:**
  1. **Although known from the very beginning, realisation constraints are ignored throughout the design phase. Furthermore, the effects of the system structure on the behavioral properties are ignored.**
  2. **The top-down approach imposes difficulties in specifying blocks whose content is ignored, and increases the risks and difficulties in decomposing sub-systems.**
- **Weaknesses of the operational approach:**
  1. **A danger of too-early design decisions is presented.**
  2. **Executing the specifications is probable not to have any performance properties.**
  3. **The transformational implementation contains all the realisation constraints and therefore is not unique but rather implementation dependent. Each individual transformation should therefore be carefully proved, prior to relying on it as assurance of the realisation correctness.**

2.6 Finite State Automata Modeling

A large variety of finite-state machines are used to specify and analyse concurrent processes (e.g. Petri nets, SPECIAL, etc.). Petri nets [31] are found exceptionally suitable for cases that involve designing and modeling of systems in which concurrent processing considerations and dynamic sequential dependencies exist. The major characteristics of Petri nets, in comparison to other models, are:

- Explicit representation of causal dependencies and independencies. When events are independent of each other, Petri nets theory introduces a non-interleaving partial order of concurrency.
- Petri net models allow description of a system in different levels of abstraction, without changing the language used in the modeling.
- Properties of the system are representable by similar means as the system is modelled. Hence, correctness proofs can be built using the same methods used for the system model construction.

The above characteristics are particularly emphasized when analysing blocking phenomenons of concurrent processing, or communication correctness.

2.6.1 Petri Nets Definition

A Petri net is defined [11] as a five tuple $< P, T, Pre, Post, M_0 >$, where:

- $P = \{p_1, .., p_m\}$ a finite set of *places*.
- $T = \{t_1, .., t_n\}$ a finite set of *transitions*.
- Pre is a mapping $T \times P \rightarrow N$ [3].
- $Post$ is a mapping $P \times T \rightarrow N$.
- M_0 is the initial marking of the net. The marking is considered as tokens occupancy of places.

A wider definition is given in [31], where maximal capacity can be assigned to places, and weights can be assigned to arcs (providing a means for controlling the firing by thresholds). In this review, all cases are limited to a unity weight.

A common graphic notation used in Petri net representation is to denote places by circles, transitions by bars, and marking by dots. Pre and Post are denoted as directed arcs.

An "execution" of the Petri net is done by changing the marking of the net via a *firing* process, which is carried out by the transitions. A transition is enabled to fire if its Pre set (also called "input places") is marked, i.e. all the members of its pre set are each marked by at least one token. The firing transfers the marking from the Pre to the Post set. The firing has three major characteristics:

1. It is *voluntary* - A transition which is enabled is not compelled to fire, yet it may fire only if it is enabled.
2. It is *instantaneous* - A firing takes zero time.

[3] N is the set of non-negative integers.

3. It is *complete* - If a transition fires, a case of a partial firing in which not all the tokens are removed or not all the tokens are placed, does not occur.

Due to the above properties, when two enabled transitions share a place in their Pre set, only one will fire. Selection of the one that fires is carried out arbitrarily.

2.6.2 Task Synchronization with Petri Nets

A task can be defined to have three states with respect to a synchronising mechanism:

1. It can be *idle* or indifferent to the synchronising mechanism.
2. It can be *waiting* at a synchronisation point.
3. It can be *active*, or using a resource after synchronisation.

When modeling a task with a Petri net, the transition depicts the active part of the task. This property is very bad for describing timing properties, since a transition is assumed to be instantaneous. In other words: when the task is active, its state in the net is undefined. The conditions for entering the active state are represented by the places.

Example [11]: When task T_1 (Figure 2.1) terminates, tasks T_2 and T_3 are activated concurrently. E_1 to E_4 are the synchronisation points (also called *exchanges*).

Yet, if a task has multiple exits, a "place" for the active part of the task must be presented, as for T_4 in Figure 2.2.

Sharing Tasks Between Processes

In some cases of two processes sharing the same task, partial order is sufficient, but when mutual exclusion is involved, total order is a must. The notation for total and partial order, using a Petri net model, is emphasised in Figure 2.3.

1. *Total order:* p_1 causes a structural conflict between t_1 and t_2. This conflict is effective in the case that p_1, p_2 and p_3 are marked by one token each. When t_1 and t_2 are executed on different processors, p_1 represents a synchronisation variable, which must be protected by a mutual exclusion mechanism.
2. *Partial Order:* When a need arises for a synchronisation variable which implements a partial order, it can be represented by a place which has a single output-transition and multiple input-transitions. In Figure 2.3 the place p_4 represents such a variable. Both t_3 and t_4 can increase its value (by firing tokens into p_4), but only one can decrease its value (in this case represented by t_5).

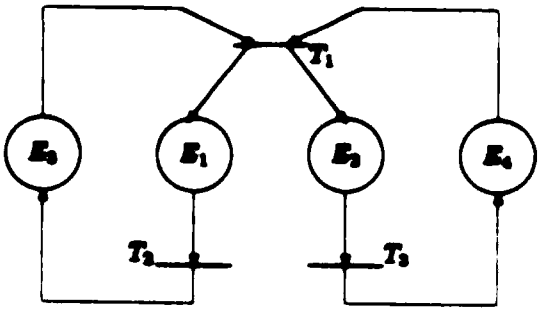


Figure 2.1: Action depicted by transitions.

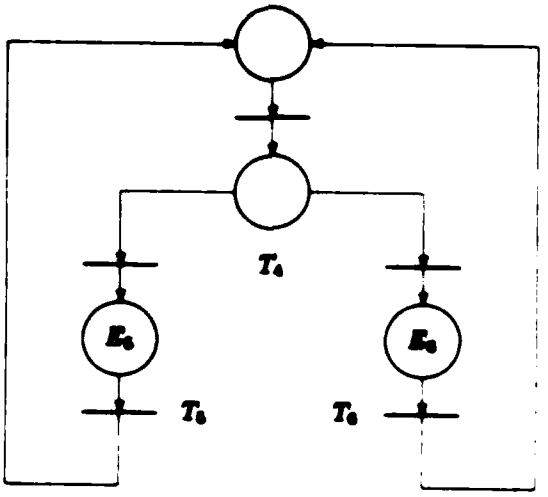


Figure 2.2: A task with multiple exits.

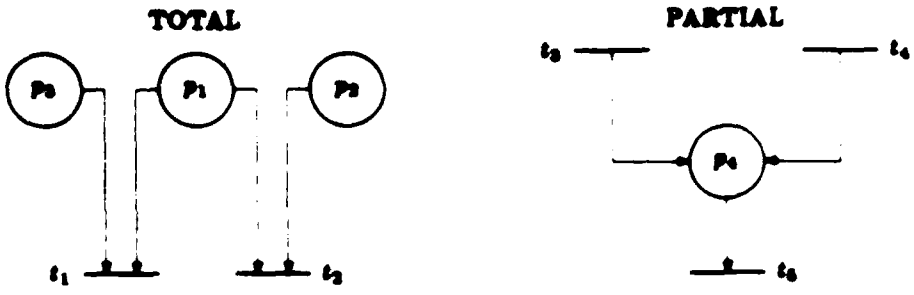


Figure 2.3: Total and partial order conflicts.

Example: Producer-Consumer [11]. 1. First, consider the synchronisation of one producer, one consumer and an n-bounded buffer (processes 1 and 3 in Figure 2.4). The tokens load on p_{pc} represents the number of free spaces in the buffer. Production beginning is represented by a firing of t_2, and a consumption beginning is represented by a firing of t_3. Since there exists no couple of transition in conflict, a partial order is sufficient.

2. Now consider the synchronisation of two producers, one consumer and an n-bounded buffer, as in Figure 2.4 : t_2 and t'_2 are in conflict over p_{pc}. Hence, a mutual exclusion mechanism is necessary, and its cost is going to be high in case the producers reside on different processors. On the other hand, the implementation of the place p_{ct} arises no problem, since it is a partial order.

Implementation: Multi-program Mono-processor

A common implementation of a synchronisation mechanism in a multi-processor environment is by using primitives as *send* and *wait*. Applying this approach directly from the net, arises a violation of the transition-firing indivisibility rule. This problem can be solved ([11]) by gathering the whole synchronisation mechanism into *one* specific task. Each of the other tasks that needs to synchronise with another, sends a synchronisation request (accompanied by the identity of the transition that has to be fired) to the synchroniser-task. The synchroniser considers the net as a database, and reacts to receiving a request by searching the appropriate transition, and when finding an enabled one by controlling the firing execution. This centralised solution has some advantages:

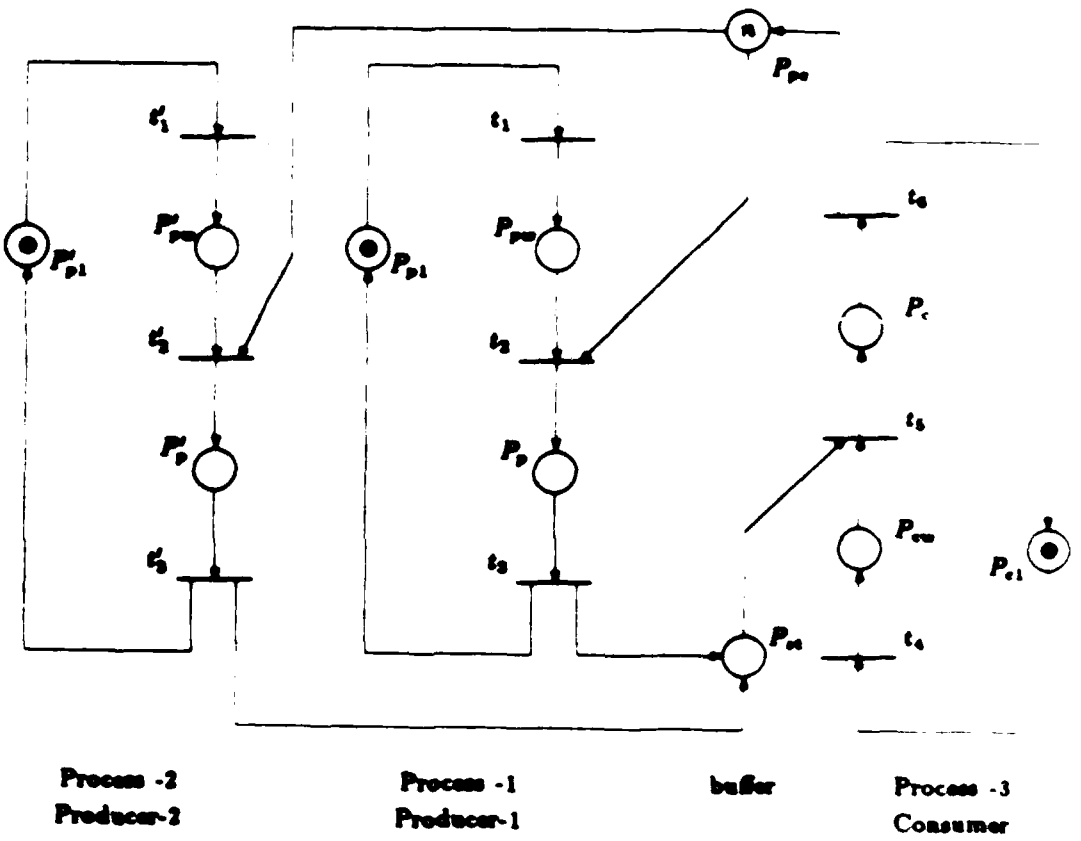


Figure 2.4: Producer/consumer: Multiprocessor model.

- A global view of the net is obtained.
- The synchronization process is derived "directly" from the net.
- Changes (when needed) are easily done, since all the process parts are located in one specific place.

Implementation: Multi-processor

Implementing the synchronization mechanism is different in the case where partial order is sufficient, from the case where a total order is necessary. In [11] the differences are described as follows.

Partial Order is sufficient: The example given for the single-producer single-consumer case shows clearly that each of the transitions t_3 and t_6 can be split into two parts. The splitting is described in Figure 2.5.

When synchroniser-1 fires t_3, it sends a message to synchroniser-2, which interprets this message as request to fire t'_3. The same approach is applied to t_6, and the result is of a non-centralized synchronization mechanism in the case of partial order.

Total Order is necessary: When applying a total-ordered synchronization mechanism, a global synchroniser picks up all the synchronization parts. In order to assure that the synchronization variables are mutually exclusive, only the global synchroniser can access them. The implementation suggested by [11] tries to capture the mutual exclusion assurance as well as to minimize the necessary communications. The solution to the two producers problem (given in Figure 2.4) is described in Figure 2.6 : t_1, P_{pw}, t_2 and t'_1, P'_{pw}, t'_2 (which represent the request - waiting - authorization) must be duplicated in the global synchroniser so it can decide upon authorization. For example: synchroniser-1 fires t_1 (sends a request message), which is received by the global synchroniser as a request for firing t_{1s}. When the global synchroniser fires t_{2s} (sends an "accept" message to synchroniser-1), it is received by synchroniser-1 as a request for firing t_2. The same scheme is applied to synchronisers 2 and 3. Hence, a simple communication is sufficient for implementing the messages required for the synchronization mechanism.

2.6.3 Time Augmented Petri Nets

As mentioned before, the Petri net classical model transitions are fired in a non-deterministic way, and do not capture the notion of time. In order to adjust this modeling method to real time programs, various modifications were applied to the classical model. An approach which assigns the execution to transitions which are time-annotated [28], is described in chapter 3. The problem with this

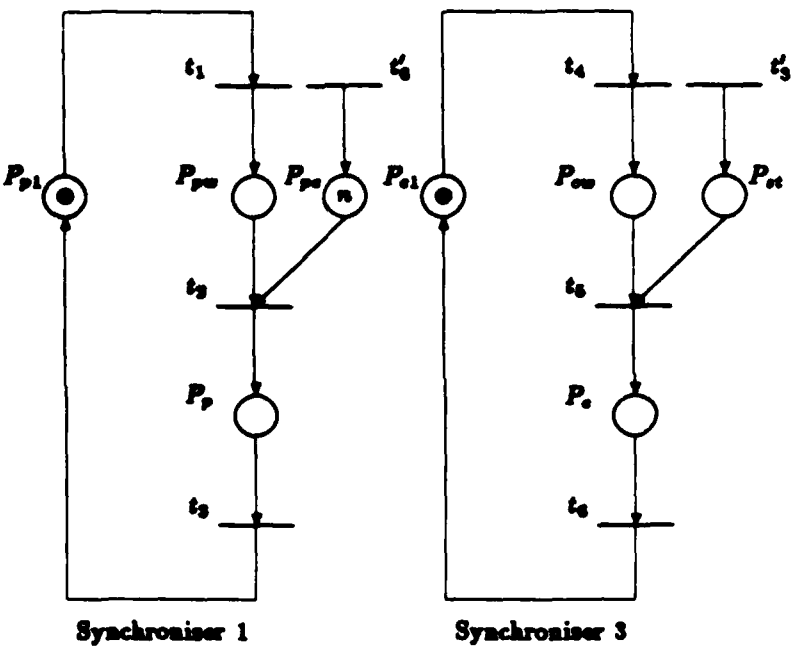


Figure 2.5: Distributed partial order synchroniser.

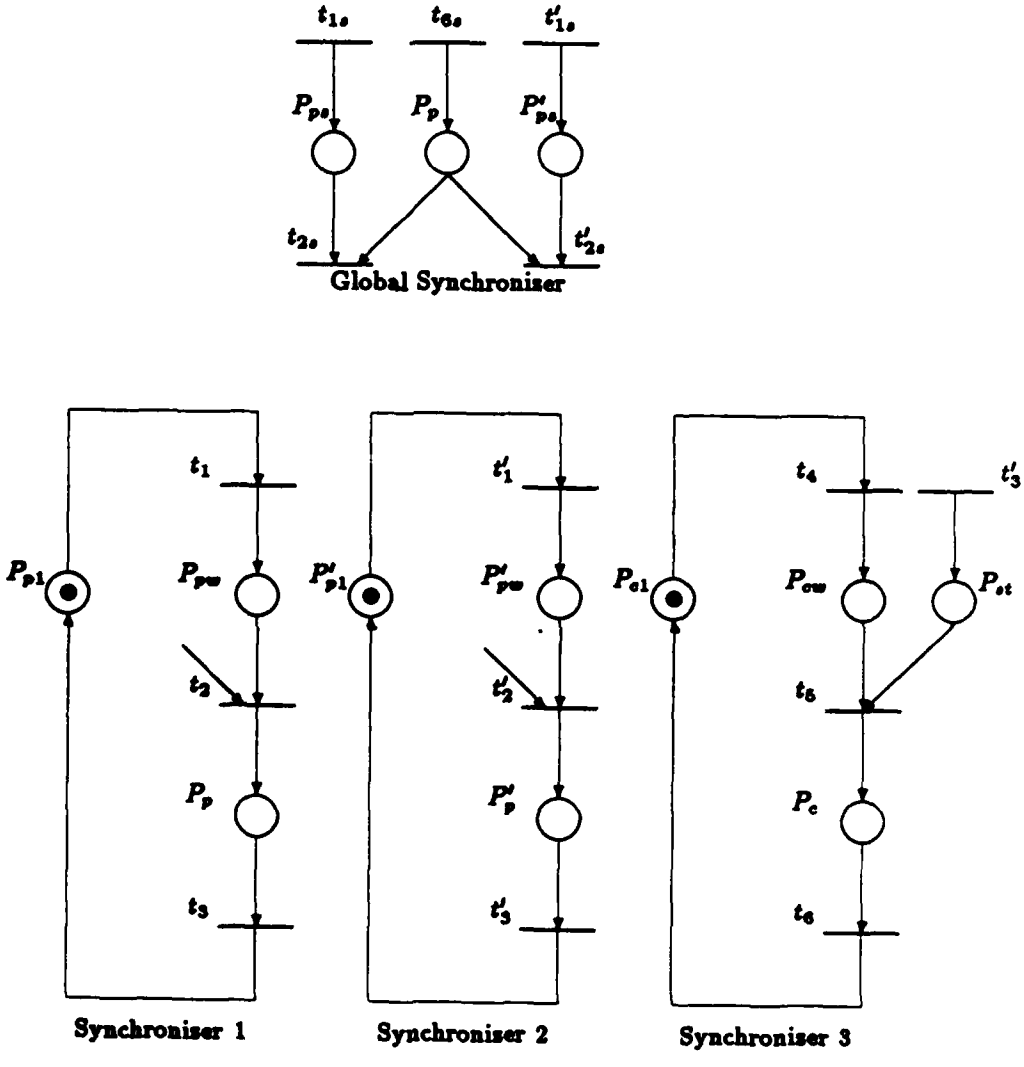


Figure 2.6: Distributed total order synchronisation.

approach is that the transitions' instantaneous nature is not satisfied completely without "inhibit" features. Another approach is to assign stochastic nature to the firing. This approach is very good for average performance-analysis, and some methods are mentioned in chapter 4 ([27,3,7]). The stochastic approach, which is very useful for verification, does not address important temporal properties which are crucially important in the design phase (e.g. scheduling, safeness with presence of time, etc.). The next paragraphs review a method [10] in which the latter issues are addressed by an augmented Petri net model, in which timing properties are assigned to places in the graph.

The Augmentation

Unlike the approaches mentioned above, in Coolahan and Rousopoulos augmentation of a classical Petri net [10], processing is represented by places, and instantaneous transitions represent start and stop of a process. Hence, a non-negative time value is assigned to each place; if the place is a "condition" then the value is zero, if it is a process then the value equals the execution time of this process. A token is ready to enable an output transition of the place it occupies, after residing on this place for the assigned time. If one transition is enabled when the token becomes ready, then this transition fires immediately. If more than one transition is enabled, then a non-deterministic selection occurs, and only one of the transitions fires immediately, while the others become disabled.

Time Driven System Model

Four *set valued functions* define the relations between nodes of the network (i.e. the directed arcs):

- $I_t(t_i)$: Transition input-function, mapping transition t_i to the set of places from which there exist arcs to t_i.
- $O_t(t_i)$: Transition output-function, mapping transition t_i to the set of places to which there exist arcs from t_i.
- similarly, place input-function ($I_p(p_i)$) and place output-function ($O_p(p_i)$) are defined.

The *cardinality* of the above sets is denoted by $|..|$.

The master timing machine, which triggers the network to be activated by a marking sequence, consists of a place denoted as p_1, connected to a transition t_1 through a loop. The following properties hold for this machine, called the *Driving Cycle*:

- The initial marking of p_1 ($m_1 = 1$) reproduces itself with a fix period T_1.
- $I_t(t_1) = p_1$.

- $p_1 \in O_t(t_1)$, and $|O_t(t_1)| > 1$.
- $I_p(p_1) = O_p(p_1) = \{t_1\}$.

The net is constructed in steps that are described below. Throughout the construction, it is guaranteed that a path, originated at the driving cycle, reaches the places or transitions which are currently added, to ensure reachability of these nodes (i.e. a liveness potential). Yet, the safeness property of the classical Petri net is affected by the reproduction property of the driving cycle, since no assurance is given to guarantee a bounded number of tokens in the net. This problem is solved by the time considerations, which enforce a bounded firing rate on the driving cycle.

Concept of Relative Firing Frequency

Two important properties are assigned to places and transitions in the net:

MRFF: (Max Relative Firing Frequency) The number of times a *transition* fires, with respect to each firing interval of the driving cycle, if all the decisions (selections of enabled transitions to fire) between the driving cycle and that transition are made "in favor" [3] of the path to that transition.

MTIAT: (Min Token Inter-Arrival Time) The shortest possible time interval between two consecutive arrivals of tokens to the relevant *place*.

An important relation is established by the above definitions: If t_i is an input transition to a place p_j, and T_1 is the driving cycle period, then

$$MTIAT(p_j) = T_1/MRFF(t_i).$$

Sub-classes of Time Driven Systems

Four sub-classes serve as construction units in the model presented in [10]:

1. **Asynchronous Systems:** For every place p_i and transition t_j the following hold:
   - $|I_p(p_i)| = |O_t(t_j)| = 1$.
   - $|O_p(p_i)| \geq 1$.

   Asynchronous systems are constrained by the following:

   1. Execution time of any process (i.e. place) cannot exceed the $MTIAT$ of this place.

[3] If a decision is pre-determined, then the ratio in which it is taken in favor of a specific path is given. If a decision is data dependent, then only assumptions or upper and lower bounds can be expressed.

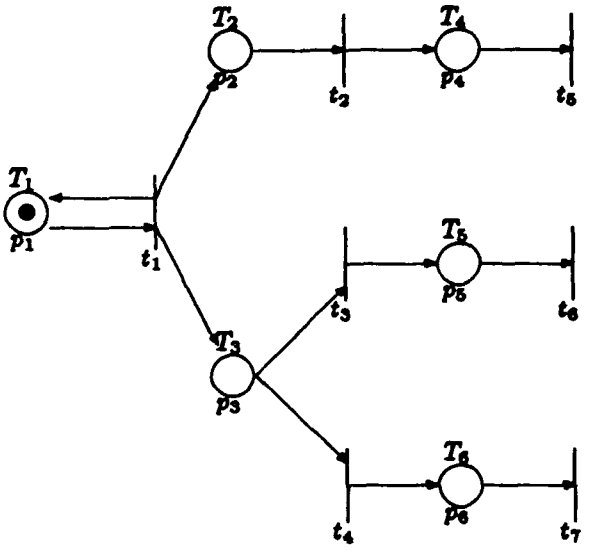


Figure 2.7: Asynchronous sub-system.

2. The cumulative execution time of any path cannot exceed any separately-stated path execution-time requirement (path latency requirement).

An example of this class is illustrated in Figure 2.7.

2. **Synchronized Systems:** In addition to asynchronous sub-systems, this sub-class includes parallel-path constructions, which consist of:

   - T: a set of transitions $\{t_i, t_f, T_p\}$, an initial transition, a final transition, and a set of zero or more path transitions.
   - P: a set of path places $\{p_p\}$.

   $P \cup T_p$ contains n (two or more) disjoint sub-sets, each representing a path from t_i to t_f. The following properties hold for the parallel constructs: [4]

   1. For t_i (the initial transition):
      - $I_t(t_i) \cap P = \{\}$.
      - $|O_t(t_i) \cap P| = n$.

[4] $\cap$ denotes set intersection. $\cup$ denotes set union. $\subset$ denotes contained relationship.

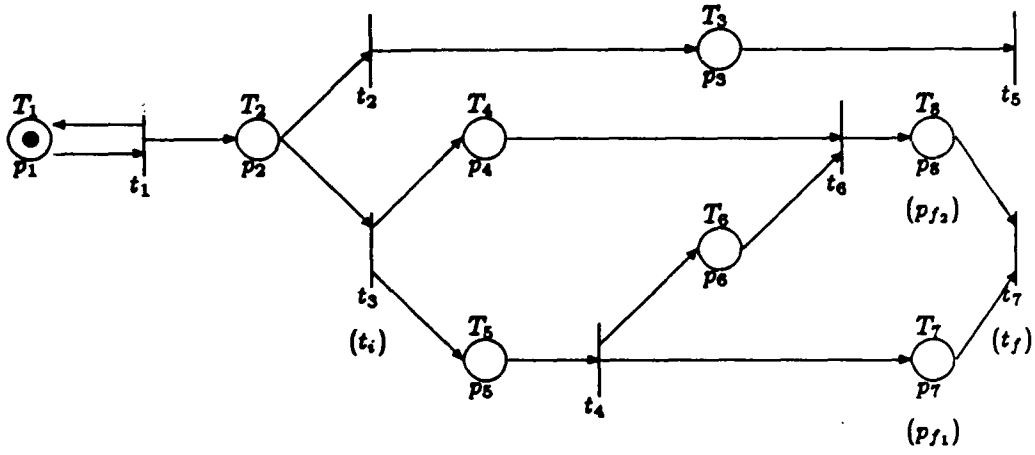


Figure 2.8: Synchronised sub-system.

2. For t_f (the final transition):
   - $I_t(t_f) \subseteq P.and|I_t(t_f)| = n$.
   - $O_t(t_f) \cap P = \{\}$.
3. For each t_p (path transition - if any):
   - $|I_t(t_p) \cap P| \geq 1$.
   - $|O_t(t_p) \cap P| \geq 1$.
4. For each p_p (path place) in P:
   - $|I_p(p_p)| = 1$ and $I_p(p_p) \subset T$.
   - $|O_p(p_p)| = 1$ and $O_p(p_p) \subset T$.

An example of this sub-class is illustrated in Figure 2.8.

An additional timing constraint is imposed on the synchronised sub-systems:

- For any set of parallel paths, delimited by t_i and t_f: the sum of execution and waiting times that a token spends at the final place (of any of the paths), must not exceed the $MTIAT$ of that place. The waiting time at the final place of a specific path is the difference between the greatest total-path-time of the set of paths, and the total-path-time of this specific path.

3. **Independent-cycle Systems:** In addition to synchronised sub-systems, this sub-class includes cycle constructions, which consist of:

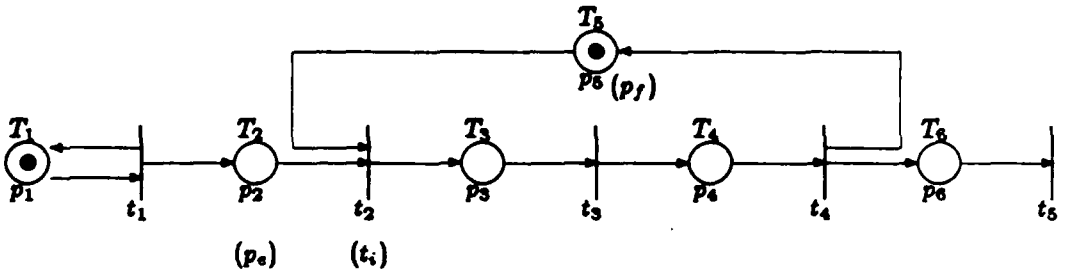


Figure 2.9: Independent-cycle sub-system.

- T: a set of transitions $\{t_i, T_p\}$, an initial transition, and a set of zero or more path transitions.
- P: a set of path places $\{p_p\}$.

$P \cup T$ represents a cyclic path from t_i to t_i. The input-places to the initial transition, are one internal to the cycle, denoted as p_f, and one external to the cycle, denoted as p_e and called the entry-place. The latter has only one input arc, and one output arc which feeds t_i. An independent cycle is characterised by the following properties:

1. For t_i (the initial transition):
   - $|I_t(t_i)| = 2$ and $|I_t(t_i) \cap P| = 1$.
   - $|O_t(t_i) \cap P| = 1$.
2. For each t_p (path transition - if any):
   - $|I_t(t_p)| = 1$ and $I_t(t_p) \subset P$.
   - $|O_t(t_p)| = 1$.
3. For each p_p (path place) in P:
   - $|I_p(p_p)| = 1$ and $I_p(p_p) \subseteq T$.
   - $|O_p(p_p)| = 1$ and $O_p(p_p) \subseteq T$.

An example of this sub-class is illustrated in Figure 2.9.

An additional timing requirement (to those of the synchronised systems) is imposed on the independent cycle:

- For any independent cycle, the cycle execution-time (from t_i back to t_i) must not exceed the $MTIAT$ of the entry place.

4. **Shared resource Systems:** In addition to all the sub-classes described so far, this sub-class permits cycles-overlap to model resource sharing. The cycles which overlap have input transitions whose firing rates are equal under all conditions, and their final places are replaced by a shared resource. This construction consists of:

   - T_p: a set of path transitions $\{t_p\}$.
   - P: a set of path places $\{p_i, p_f, p_p\}$.

   The following properties characterise the shared-resource construct:

   1. For p_i (the initial place):
      - $|I_p(p_i)| = n$, one input transition from each of the n cycles.
      - If $p_i \neq p_f$ then $|O_p(p_i)| = 1$ and $O_p(p_i) \not\subset T$.
   2. For p_f (the final place):
      - $|O_p(p_f)| = n$, each output transition being an input transition to one of the cycles.
      - The initial marking is *one* ready token.
      - If $p_i \neq p_f$ then $|I_p(p_f)| = 1$ and $I_p(p_f) \subseteq T$.
   3. For each p_p (path place - if any) in P:
      - $|I_p(p_p)| = 1$ and $I_p(p_p) \subset T$.
      - $|O_p(p_p)| = 1$ and $O_p(p_p) \subset T$.
   4. For each t_p (path transition):
      - $I_t(t_p) \subset P$.
      - $O_t(t_p) \subset P$.

   An example of this sub-class is illustrated in Figure 2.10. The initial marking in p_f is the semaphore that controls the mutual-exclusion of the resource.

   The constraint that was mentioned while introducing this construct, may be applied as a restriction to the construction process; ensuring that all firing frequencies of all the input transitions to the entry-places are equal must be applied before continuing the construction. Additional timing constraint for the shared-resource constructs is

$$T_{ej} + \sum_{k=1,\neq j}^{n}(T_{ck}) \leq T_d,$$

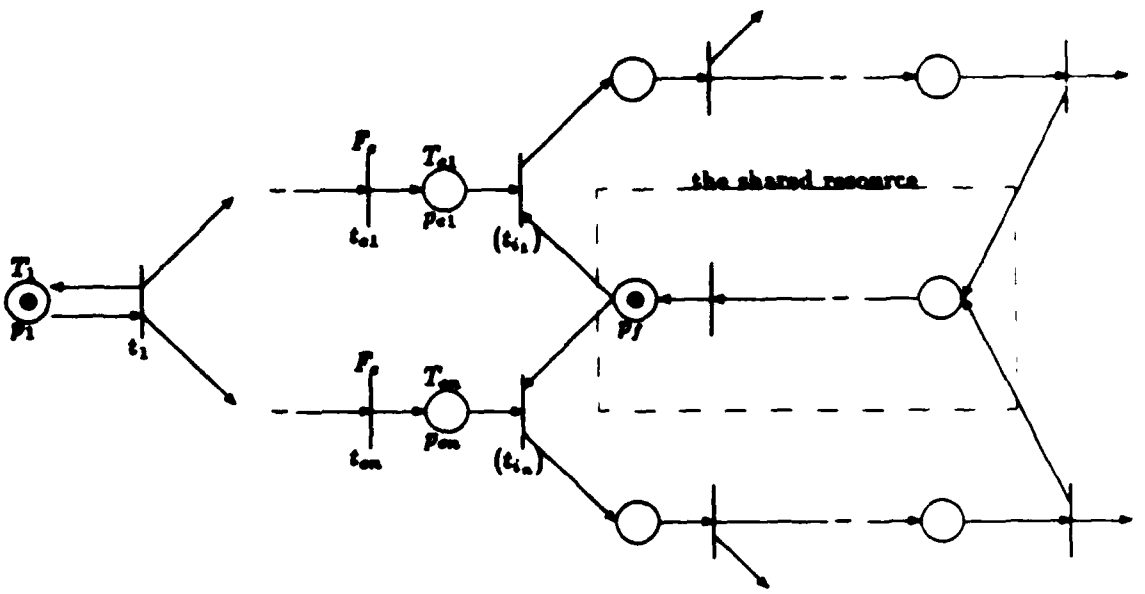


Figure 2.10: Shared-resource sub-system.

where:

- T_{ej} - Execution time of entry-place p_{ej}.
- T_{eh} - (or T_{ej}) Execution time of all places in the cycle, which are activated as a consequence of the input provided by p_{eh} (or p_{ej}).
- T_d - The $MTIAT$ of each of the n entry places.

The above constraint ensures that p_{ej} is safe in presence of time.

Net Construction Methodology

The construction of the augmented Petri net, is represented in [10] in an algorithmic way, in which all the above constraints are taken into consideration. The method consists of the following steps:

1. Construct a driving cycle $(p_1 - T_1, t_1)$.
2. Add places to the driving cycle's transition (t_1) as the system to be modeled requires, such that each of the added places has:
   - a single input arc,
   - a finite execution time.
3. To each of the places which does not have an output so far, one or more of the following constructs are added as an output (according to what the system to be modeled requires):
   - A single transition with exactly one input arc.

- A complete parallel synchronised-path construct.
- A transition with multiple inputs, which completes a synchronised-path when added to the sub-net that is already constructed.
- An independent-cycle construct.
- A cycle which forms a part of a shared-resource construct, guaranteeing that all entry-places fire at the same frequency.

4. Add output places (as in step 2) to each transition which has no output-place so far.

5. Repeat steps [3-4] until the model is complete.

Derivation of Max Relative Firing Frequency

The firing-frequencies of transitions, expressed in terms of the frequency of the driving cycle, can be considered as mapping the transition to this property's domain. Using the above mapping, a model of a system can reflect either consistency or inconsistency.

- A system is considered *consistent* IFF there exists a positive-integer mapping to each transition, such that at every place p_i, the sum of the integers on $I_p(p_i)$ equals the sum of the integers on $O_p(p_i)$.
- A system is considered *inconsistent* if it either produces an infinite number of tokens, or consumes the tokens and stops.

In practice, the integer assignment to each transition gives a possible *relative firing-frequency* to the transition. Consistency can be detected by solving the local-balance equations described above, and finding a solution with no contradictions. As can be seen in the model's description, the firing-frequency of each transition is rooted in the driving cycle. An emphasis should also be put on the difference between this approach and the classical Petri net approach. The transitions here fire immediately after being enabled, while the firing in the classical approach is non-deterministic. Therefore, here the firing-frequency of each transition can either be computed definitely (all decisions in the path are predetermined), or bounds can be set (in case there exists a data dependent decision in the path).

Proving Safeness in Presence of Time

The construction methodology given above guarantees the reachability of the places in the network. Yet, if this network is interpreted in the classical Petri net orientation, the master timing mechanism of the model – the driving cycle – does not preserve safeness property, by allowing "exponential explosion" of the tokens population in the network. Therefore, in [10] the property "safe in

the presence of time" is introduced, and the timing restrictions for this property are derived, to hold for each of the net constructs introduced above. A brief summary of the restrictions is given below.

Safeness for simple places. A "simple place" is a place p_i with $|I_p(p_i)| = 1$ and $|O_p(p_i)| \geq 1$, such that each transition t_k in its output set satisfies $|I_t(t_k)| = 1$. In [10] it is proved that such a simple place is proved to be safe in presence of time in the worst case IFF

$$T_i \leq T_1/F_j$$

where: T_i is the execution-time of this place, T_1 is the cycle-time of the driving cycle, and F_j is the $MRFF$ of the input-transition to this place.

Safeness for final places in parallel path. The final places p_{fi} ($i = 1..n$), of a synchronised parallel construct with n parallel paths, are each found safe in presence of time in the worst case, IFF for each of them

$$\forall j \in 1..n : P_j - P_i \leq (T_1/F_{fi}) - T_{fi}$$

where: P_j is the total path time (execution + waiting) of path "j", F_{fi} is the $MRFF$ of the input transition to p_{fi}, and T_{fi} is the execution time of p_{fi} itself.

Safeness for entry places of independent cycles. Cycle-time (T_c) is the time interval from firing of t_i to "token ready" in p_f. The execution time of the entry place is denoted as T_e, and the $MRFF$ of the input transition to entry place is denoted as F_e. In [10] it is proved that an entry place to an independent cycle construct is safe in presence of time IFF

$$T_e \leq T_1/F_e \text{ and } T_c \leq T_1/F_e.$$

An outcome of this result is that if p_e is safe in presence of time, then waiting time for *all* tokens arriving to this place is *zero*.

Safeness for places in a shared-resource. A shared cycle time T_{ch}, for a cycle containing t_{ih} in a shared resource construct, is the time that elapses from firing of t_{ih} until the token arrives at p_f. It is proved (in [10]) that given that all the entry places receive from identical $MRFF$s, and the $MTIAT$ of the entry places is their execution time divided by the $MRFF$, then each entry place p_{ej} (to a shared resource construct) is safe in the presence of time IFF

$$T_{ej} + \sum_{h=1}^{n} T_{ch} \leq T_{ej} + T_1/F_e.$$

For all the above constructs' types the regularity of the net timing is preserved.

Conclusion

Coolahan and Roussopoulos examine the method they suggest in [10]. The weakness points they find in their approach are

- The results are very sensitive to changes in process or path execution time.
- An alteration construct is needed.
- The restriction on a shared resource (limited only to the case of identical firing rates) is too strong.

The advantages they find in it are:

- The ability to formally and explicitly express the timing constraints.
- The automatability of the method.
- The ability to verify timing properties in the design phase.

Two other deficiencies should be considered. First, the absence of performance analysis tools, does not allow the examination of timing problems rather than safeness (e.g. bottle-neck). Second, the restrictions imposed in the construction of the net, narrows significantly the cases that can be treated.

Chapter 3

Development in A Real Time Environment

3.1 General

When the design phase is complete, and a verified solution for a given problem exists, the problems which concern the implementation of the solution arise. Some software engineering research tried to evaluate the approach adopted in projects using various criteria, as the changes installed in a program during its life cycle [8], but only few works tried to evaluate the approach from the timing point of view. As emphasised before, real time software performance depends highly on the implementation, and therefore the definition of the environment is of crucial importance. The development environment includes the processor used, the operating system, the language and its compiler and run time libraries, the network structure in which a distributed computation takes place, and special aids used during the implementation phase. Some attempts have been made to allow proper identification of this environment (e.g. [20]), and experience was summarised into recommendations (e.g. [1]), but probably the most important events in real time software implementation in the last decade are:

- The standardisation of the Ada programming language by the U.S. Department of Defense.
- The adoption of MASCOT by the U.K. Ministry of Defense.

This chapter tries to highlight some important aspects concerning real time software implementation; Starting with the influence of the design approach adopted, reviewing an automated tool which translates a model to a program, emphasising the importance of the programming discipline, and concluding with two important aspects concerning programming with Ada.

3.2 Approaches

Specifying a system in a conventional approach treats the system as a "black box", describing *what* but not *how*. In other words, when we start the realisation phase (adopting the conventional approach) the external behavior is well defined, while the internal structure is not. Top down discipline is maintained throughout the whole development phase, which means that implementation constraints, although known from the very beginning, are intentionally ignored. Some trials have been made in information processing projects (as the Jackson development approach) to find better solutions. The PAISLey project uses a new approach in real time system development, in which these ideas are organised into a strategy called *operational approach* [35]. The main idea is that external behavior and internal structure may interleave. Yet, operational approach separates the problem-oriented structure of the operational specifications from pure implementation considerations. The operational specifications themselves are written in an operational specification language, which is executable and prevents ambiguities. Transformational implementation would therefore guarantee correctness of the produced code. Although this approach seems to lead to a very easy implementation phase, the problems of stringent timing constraints do not disappear – they can be found in the generation of the transformational implementation.

3.3 From a Model to a Program

In chapter 2 of this paper, the Petri net models were introduced. For systems which contain concurrent processing considerations and dynamic sequential dependencies, this modeling technique seems exceptionally suited. Nelson et al [28] introduce a method of translating a Petri net model into a procedural language program, and this method is reviewed in this section.

3.3.1 Annotated Petri Nets

The commonly used Petri nets are extended by means of *annotations* and *initial considerations* that provide processing content and external dependencies to the firing of the net.

1. Actions that do not relate to the net itself (e.g. applying a function to a specific data structure, firing another net, calling a procedure) are assigned to *transitions* with a corresponding annotation.

2. Boolean expressions that express external dependencies are assigned through an appropriate annotation to *output arcs* of a transition.

3. An integer selector may be assigned to a *transition*. When this transition fires, at most one of its output places will be marked.

4. Two special types of transitions are allowed to have only input or output arcs: the *initial* and the *terminal* transitions.

3.3.2 The Method of Translation

Each node in the network (either a transition or a place) is related to a specific *template* of statements. The template content depends on the node's type, its fan-out, and the annotation assigned to it. combining these templates results in a program in a procedural language called XL/1, and this program is then (via an automated process) translated to PL/I or PL/S.

3.3.3 Multiprocessor Environment

Petri net theory does not force a fireable transition to fire, but if a transition fires the firing is complete and consumes zero time. Implementing a transition firing with a program imposes a non-instantaneous firing. When the environment is of a multiprocessor system, a mutual exclusion device must protect the firing, since more than one processor may be focused on particular input places at a specific time. Therefore, the authors provide a semaphore-like *locking* mechanism, and an *atomic* operation which adjusts (increment or decrement) the number of tokens in places. All the places that are associated with a specific lock are assigned (according to given rules) to a lock-set, which governs their access to the lock.

3.3.4 Conclusion

Although the mechanism proposed in [28] is not optimised, especially for hard real time applications, it provides a very good tool for the design verification. If further work will be done, architectural optimisation may be performed by the XL/1 automaton, imposing the constraints and the implementation dependent properties of the system.

3.4 Implementation Discipline

The complexity of programming increases when we step from sequential to multiprogramming, and it increases further when we apply real time programming. A set of *concepts* (for reasoning) and a set of *facilities* (for description) is added in each step. Adding synchronisation signals and mutual exclusion devices to sequential programming allows us to use multiprogramming, and adding to it execution speed allows dealing with real time programming. But the complexity of reasoning increases by a new dimension. In [34] a summary of all the needs for real time programming discipline is given and analysed.

3.4.1 Making a Real Time Program Manageable

In order to make a real time program manageable, Wirth suggests the following recipe:

1. First formulate the entire program without reliance on execution times. All necessary synchronisation signals should be provided explicitly.
2. If the machinery to be used does not provide some of the necessary synchronisation signals, derive analytically the timing constraints for each, and allow its absence.
3. Check whether the constraints are satisfied by the computer system.

3.4.2 Synchronization Discipline

In a distributed environment, processes commonly synchronise using *signals* and *semaphores*. A semaphore is equivalent to a signal with associated memory. Sending an unconditional signal, when no process is waiting, may lead to an untraceable system crash. This mistake is commonly made due to assumptions taken over the computation speed of the different processes. Hence, declaring *await(s)* as a necessary precondition of *send(s)* is recommended, when occurances as below might exist:

P1: ...S1; *send*(*s*); ...
P2: ...S2; *await*(*s*); ...

3.4.3 Language and System Requirements

The form and structure of a real time programming language are as important as in regular distributed programming. No additional structural concepts are needed for a real time programming language:

1. A notational unit for describing processes (themselves sequentially executed) that can be executed concurrently, and noninterruptably.
2. A collection of shared variables and their operators.
3. An object to trigger communication after waiting (signals).

Yet, an additional feature is needed:

- A facility which provides accurate execution time bounds, as an additional part of an existing compiler. If the compilation is straightforward (i.e. no optimisation is performed), the use of a simple execution time-table of the statements is possible.

An important recommended concept is the *coherency* of logical processes (each one is implemented as if it owns a private processor). Two features that are achieved applying this concept are the simple logic foundation of the program, and the maximal degree of freedom left for choosing processor sharing strategy.

When applying processor sharing policy, the static time bounds (prepared in a straightforward time-table) do not hold; the processor may be attached to another process for an unknown "time slice".

3.4.4 High Level Language and Processor Sharing

Most of the uncertainties in execution time are due to tasks performed by another process. The author uses the notation "doio" (as in DO I/O) for the following sequence:

...send(initiation);wait(completion)...

and applies a "hidden" delay statement behind each "doio" statement. The delay is derived from the execution performances and the strategy chosen for the processor sharing (priority plays a major role here).

3.4.5 Recommended Discipline for Real Time Programming

1. Time dependent program parts executed externally (by a device process), should be restricted to noninterruptable execution mode.
2. Execution time of the above program parts is determined statically.
3. Each "doio" is assumed to have a hidden delay, derived as stated above.

This discipline may lead to high time consuming delays, and the proposed solution is a *priority interrupt system*, which requires adherence to the following constraints:

1. Every device process P_i is *cyclic*, consisting of a statements sequence S_i, and the "doio" represents the waiting for device completion.
2. $t_i = T(S_i) + T(doio_i)$. The cycle time of P_i, at any priority level, is considerably greater than that of all other processes at higher priority.
3. The ratio
$$r_i = \frac{T(S_i)}{T(S_i) + T(doio_i)}$$
over any cycle is very small ($\ll 1$).
4. Each signal emitted by a device must be awaited by single (regular) process only.

5. A device process must never *itself* invalidate the condition associated with the signal it emitted.

3.5 The Ada Programming Language

The Ada programming language [2] was designed as a common language for programming large scale and real time systems. The Ada programming language has introduced many high level facilities, but in this review only one innovation is examined. The tasking facilities have always been a part of the operating system, rather than the programming language, until Ada has been introduced. Furthermore, synchronization of these facilities has used primitives provided by the operating system, invoked usually as "system calls" from a very low level part of the program. The Ada programming language includes both the tasking facilities, and the synchronization tools as a part of the language. This feature allows the programmer to concentrate on parallel system design and to ignore inter-task synchronization and communication details.

Parallel processes are called *tasks* in the Ada language. Each task may have some *entries*, which are called from other tasks. Two tasks interact by first synchronising, then exchanging information and finally continuing their individual activities. This synchronized meeting to exchange information is called *rendezvous*. This concept is based on Hoare's CSP proposal [19] for concurrent programming, and Ada is the first language that has adopted it. Hence, this is the only experience in using this concept, and recent works have shown some interesting aspects in implementing it.

Two important issues concerning the rendezvous are reviewed in this section: the implementation of the concept in the compiler level, and the implementation of the concept in the program level. As stated below, both may lead to inefficiencies and undesirable effects.

3.5.1 Implementing Tasking Facilities

Three ways of implementing the rendezvous concept in the compiler level are examined in [14]. The paper also examines results of the three implementations in PASCAL to validate the analysis. The mechanism and the implementations are described below.

Assumptions and Description of the Mechanism

The assumptions taken in [14] are:

1. The Ada kernel is implemented as a set of primitives. An exact copy of this set resides on the private memory of each processor which participates in executing the concurrent program. The interaction between tasks that

these primitives allow, is independent of the physical allocation of the tasks.

2. The executable code for each processor resides on its private memory.

3. Some data constructs are located in the system shared memory.

4. Each task descriptor consists of two parts: global and local task descriptors. The local task descriptor resides on the private memory of the processor which runs this task. It contains all the information required to run this task concurrently with other tasks resident on the same processor (i.e state-word, status field, task priority, scheduling time, links to other local task descriptors, a link to its own global task descriptor). The global task descriptor resides on the system shared memory. It contains all the information required to allow interaction between tasks that are allocated in different processors (i.e. processor Id, a lock variable for mutual exclusion, status field, pointer to actual parameters list, a set of addresses of entries, a set of queues - one for each entry, a stack of invoking tasks Id's, a link to its own local task descriptor).

5. Each processor can send interrupt signals to any other processor.

The task management mechanism suggested in [14] is:

1. When a task invokes the kernel to interact with another task, the invoked primitive (executed by the processor on which the calling task resides) checks the global descriptor of the called task.

2. IF the called task resides on the same processor as the calling one, THEN all the required information concerning its status is available in the local descriptor.

3. IF the called task resides on a different processor, THEN the primitive which has been invoked sends an interrupt request to that processor, specifying the calling processor, the called task, and the requested operation. The interrupt service procedure will then invoke the kernel primitive which correspond to the requested operation, to complete the interaction.

Implementation of the Mechanism

Three possible implementations for the above mechanism are described below. The comparison criteria are minimising system overhead, and the task blocking time.

"Server" Rendesvous is the first possibility to implement the mechanism. In this implementation the calling task remains suspended until the called task executes the *accept* body (see [2] for description of the accept statement). This implementation has advantages and deficiencies which are listed below.

1. A single copy of the accept body is sufficient, and it should be stored in the private memory of the accepting processor.

2. In order to complete the rendesvous, the scheduler is invoked (possibly a context switch occurs) *twice* in the case that the entry calling execution precedes the accept execution, and *three times* otherwise. *One* or *two* inter-processor interrupt signals are required, and *two* or *four* scheduling operations (respectively) are necessary, if the interacting tasks are running on different processors.

3. Parameter passing may be carried out through the shared memory.

"Procedural Call" rendesvous is another possible implementation. Here the accept body is always executed by the calling task. This approach has the following properties:

1. Accessibility of the accept body can be obtained either by keeping an exact copy of the accept body on each private memory of a processor that runs the calling task, or by storing the accept body in the shared memory. The shared memory solution is exactly the same as the "server" solution (communication-wise). The replication solution may be ineffective or impossible if a resource needed for the accept body is only available to a particular processor.

2. No special mechanism of parameter passing is needed, since the caller executes the accept body in its thread of control.

"Order of Arrival" rendevous is a solution provided by the authors of this paper ([14]), and it reduces the scheduling points required. Here, the accept body is executed as a part of the thread of control of the last task which joins the rendesvous. The properties of this approach are:

1. In the case of a mono-processor system only *two* scheduling points are needed.

2. In the case of tightly coupled multi-processor system, *one* inter-processor interrupt signal and *two* scheduling operations are needed to complete a rendesvous.

3. The same resource allocation difficulties that were introduced in the "procedural call" approach exist here.

The differences between the three approaches emphasise the significance of the compiler-level implementation, for the timing performances as well as for the communication economy of a system.

3.5.2 The Tendency to Poll

Experience in using the rendesvous concept when programming in Ada, is summarised in [15], pointing out a tendency to apply polling policy, which is usually (but not always) undesirable because it is wasteful of system resources. A review of this findings and the suggestions provided in the above paper are given below. But first, four assumptions concerning the rendesvous mechanism must be taken into account:

1. Two tasks, A and B, rendesvous at entry E of B, when A calls entry E, and the entry call is *accepted* by B.
2. IF A calls the entry call before B is ready to accept the entry call, THEN A waits until B is ready.
3. IF B is ready to accept, THEN it must wait until some task issues that entry call.
4. Calls to a specific entry are executed in FIFO order.

The Rendevous Statements

Two types of rendesvous statements are permitted by MIL-STD-1815A [2], the selective wait statement and the conditional entry call.

1. *The Selective Wait Statement* [1].

```
select
      [when cond ⇒] selective_wait_alternative
{or
      [when cond ⇒] selective_wait_alternative}
[else
      sequence_of_statements ]
end select;
```

The selective wait statement is used for waiting and selection from one or more alternatives. A selective_wait_alternative is restricted to the following:

- A *delay* or an *accept* statement, followed by a sequence of statements.
- The *terminate* alternative.

[1] [..] - stands for optional statement. {..} - stands for zero or more times.

The selective_wait_alternative is non-deterministically selected from the set of "open" accept_alternatives for which a rendevous is immediately possible (if the set is nonempty). The approach is adopted from [19]. A delay_alternative is selected if no accept_alternative can be selected before the specified delay has elapsed. If no selective_wait_alternative can be selected, the else-part (if nonempty) is executed. If no accept_alternative is immediately possible, and there is no else-part, then the task *waits* until such an alternative will become open.

2. *The Conditional Entry Call.*

```
select
        entry_call_statement [ sequence_of_statements ]
else
        sequence_of_statements
end select;
```

Although the syntax is quite similar to that of the selective wait statement, they are semantically opposite: the selective wait is used in accepting entry_calls, while the conditional entry call is used for making entry_calls. IF the rendesvous is not immediately possible, THEN the entry_call is canceled, and the else-part is executed.

Polling

Polling is characterised by a task *actively and repeatedly* checking for an occurance of an event that originates externally to the task. The paper ([15]) distinguishes two types of polling:

1. Task A *rendevous polls* with task B (with respect to entry E) IF the rendesvous can be preceded by an unbounded number of attempts by A. (Attempt is defined as an unsuccessful entry call OR a failure to select an accept_alternative in a select statement.)

2. Task A *information polls* with task B (with respect to entry E) IF A and B can rendesvous an unbounded number of times before information is exchanged.

A *busy waiting* situation is identified if between rendesvous attempts no computational progress is achieved. The polling is usually wasteful of resources – it simply burns up CPU cycles. Furthermore, it may unnecessarily load the communication network very heavily. Another dangerous situation happens when the calling and the called tasks both symmetrically loop over a

selective_wait_statement and a conditional_entry_call with an else-part: the rendesvous may never occur! Yet, sometimes polling is desirable: when non-polling may result in such an additional overhead that might violate real time constraints. But these cases are very rare, and should be definitely identified and justified.

Bias Towards Rendevous Polling

The tendency to poll when implementing the Ada rendesvous is encouraged by the following:

- Lack of some facilities.
- Restrictions imposed by Ada.
- Presence of some facilities.

Four encouragements for polling are given below.

Conditional Entry Call should be used with care, since it may lead to unnecessary polling. Consider the poor example given in [2] paragraph 9.7.2.7:

```
procedure SPIN(R : RESOURCE) is
begin
        loop
            select
                R.SEIZE ;
                return ;
            else
                null ; – busy waiting
            end select ;
        end loop ;
end ;
```

The "busy waiting" is really unnecessary.

Handling an Entry - Family is generally expressed as a polling loop. For example, let X be an entry-family that have N members, declared as *entry* X(Y), while Y is of subtype integer that ranges from 1 to N. The skeleton of the program that accepts call for the entry-family usually looks like:

```
loop
    for I ∈ Y: loop
```

```
                select
                      accept X(I) do ... end ;
                else
                      null ;
                end select ;
        end loop ;
end loop ;
```

This select statement polls unnecessarily, and solution may be given either by replacing the "for" statement by an N specific ORed accept_statements with no else-part (only for a small known N !), or by a different design which makes a use of the entry family indices.

Restrictions on Selective-Wait-Statement of two types are imposed.

1. Not allowing a *when* condition followed by a sequence of *non-tasking* statements as a *select alternative*, which is needed as follows:

```
loop– illegal example
     select
     .
     .
     or
       when cond => A := B ; – illegal !!
     end select ;
end loop ;
```

The above implementation is illegal since only accept/ delay/ terminate are allowed as alternatives. The use of an *else - if* statement to replace the illegal statement is wrong, since it would be executed to no effect (again and again) when any feasible alternative is absent. Hence,

```
loop– better solution
     select
     .
     .
     or
       delay 0.0 ; – zero delay is legal
       A := B ;
     end select ;
end loop ;
```

2. The lack of *selective_call* statement and not allowing any *entry_call* as a *select alternative*, which are needed as:

```
loop– illegal form
     select
       call_entry X.E ;
     or
       call_entry Y.F ; – illegal !!
     end select ;
end loop ;
```

Replacing the illegal "or" part by:

```
else
    select
          call_entry Y.F ;
    else
          null ;
    end select ;
```

is wrong for two reasons: it gives preference to X.E and the inner select polls again. Another problem arises when we need an entry call as a select alternative:

```
loop– illegal form
     select
       when B => accept ...
     or
     .
     .
     or
       when C => call_entry Y.F ; – illegal !!
     end select ;
end loop ;
```

Since the above program is illegal, one is tempted to replace the "or when C ..." with:

```
else
    if C then
      select
```

```
    call_entry Y.F ;
  else
  null ;
  end select ;
end if ;
```

This replacement may lead to a dangerous situation, especially when X and Y are symmetrical, since a simultaneous polling becomes possible, and the rendesvous may never occur.

The "else" clause in a selective-wait-statement is the highest temptation to poll. Therefore a careful examination should be taken: IF the alternative action is not really a part of the task, THEN it should be encapsulated in another task, which could result in the elimination of the polling. A good design approach separates the tasks functionally - one task for each function.

Suggestions

The authors provide some suggestions to changes in the Ada programming language, but they agree that "it is likely that the Ada programming language will not be modified, at least not in the near future". Three major principles pointed out by this paper should be adopted:

1. Use a *delay* alternative with a zero delay, to allow a *when* condition followed by *non-tasking* statements as a select alternative.
2. Separate the tasks functionally - *dedicate one task for each function.*
3. Take a lot of care in the program construction to avoid unnecessary polling. For the cases in which polling is a better solution than others, justify it carefully.

Chapter 4

Verification and Validation of Real Time Software

4.1 General

Verification of programs (i.e. proving that a program meets its specification), may be done in three major methods:

- Using an *axiomatic approach* to infer mathematical and logical assertions which describe the control and data states.
- Using *Box-Line* or *Graph* properties to obtain the required proof.
- *Testing* the program for the relevant inputs it may meet when executed in its target plant.

The axiomatic approach is commonly preferable, because assertions can be communicated to computers via compilers, and then manipulated by simplification procedures. Due to this, proof systems for many design and development approaches have been introduced. Axiomatic proof systems have been introduced for distributed systems [29], and for CSP programs [6]. Yet, these proof systems do not deal with timing properties, and provide no means for real-time verification. The complexity of verification grows significantly when the implementation is required to be distributed.

Most of past works based their proofs (concerning time properties) on queueing theory [23], proving average performence criteria, since the timing characteristics of inputs to a real time program are stochastic by nature. In addition, these works supported the conclusions by tests which cover the inputs' range.

An example of this approach is described by G. Anderson [5], combining five methodologies for evaluating performance properties:

1. Characterisation of work load to the proposed system.
2. Creation of an approximate queueing model for the system, and evaluating average performance properties.
3. Identification and preparation of hardware tools, to allow measurements in the real system.
4. Development of a load-simulator, to allow testing under a controlled load.
5. Modeling the system with a detailed simulator, which allows bottle-necks identification and answers to "what if" questions.

Anderson's results have shown good match between expected and achieved values (11% in response time, 2% in CPU utilisation), yet most of his assumptions were based on previous experience, which is always needed but rarely found.

In this chapter, various methods of programs validation techniques will be reviewed, starting with testing real time performances, followed by analysis and proof methods for real time properties, and finally examining the use of simulations for system validation.

4.2 Testing Real Time Properties of Programs

U. Voges and J. R. Taylor [30] review many testing approaches and procedures, all sharing the same goals: proving that the system under test is free of errors, and obtaining (when it is possible) some figures about the system's reliability.

4.2.1 Systematic Testing Methods

Testing Coverage

Testing a system thoroughly means testing it with all possible combinations of its inputs. In exercising input sequences of a distributed system, relative changes within an input sequence are very important as well (e.g. synchronisation problems), a property which may lead to an enormous sequence of inputs for such a test. An early approach, suggested a design criteria of asynchronous reproducibility of output (for a set of inputs, the same output will be produced, regardless of speed differences or time intervals at which the inputs are delivered). Although this is a desired goal, sometimes it is not achieveable, especially dealing with real time systems in which a deadline criteria should be met. Yet, adopting this approach in the design even partially, reduces significantly the amount of required input sequences.

"Glass-Box" Testing Methods

This method is applicable especially in module-testing level. It consists of analysing the module reachable paths, comparing the calculated path predicates with the specification, followed by symbolic executions of the tested module. Its major disadvantages are the ignoring of dependencies between modules, the inability to deal with run time changes of control logic, and the inability to pinpoint a missing path.

"Black-Box" Testing Methods

In this test method no inside look is concerned. A test should be performed both positively – a functional test with inputs chosen according to the specification, and negatively – reaction to abnormal and unspecified events.

Probe Effects

The availability of "probing points" in the real system is usually limited. In order to trace control flow or data values, additional probing statements are inserted to the program. Therefore, changes regarding the environment to be met in the real operating mode are introduced. This effect is of extreme importance when dealing with a hard real time environment, and in many cases disqualifies this testing method.

Example: SADAT

SADAT [33] is an automated test tool, which supports testing of a single FORTRAN module. SADAT performs the following test procedures:

- *Static Analysis* – Generates the program control graph, in which sequential parts are represented as nodes and the arcs are an interpretation of decision to decision (*d–d*) paths. This analysis may detect unreachable statements and errors in control flow that the compiler failed to detect.
- *Test Case Generation* – Produces a minimal subset that ensures at least one execution of each d-d path.
- *Path Predicate Calculation* – Produces the path predicate for every path in the module, and runs a symbolic execution.
- *Dynamic Analysis* – A control statement (in the form of a subroutine call) is inserted in each d-d path, allowing accumulation of number of executions for each node. This output can be used to track a dynamically "dead" code, optimisation of the most frequently executed parts, and for identification of additional test cases that are required.

SADAT major disadvantages are the lack of distributed and real time properties testing. Dealing with a single module does not allow any concurrency and parallelism, and since no deadline analysis is performed, no critical timing problems can be pin-pointed.

Example: TAS

Ferranti Computer Systems Ltd. (UK) developed [13] a complete environment they use for software development and validation of real time software, using MASCOT (Modular Approach to Software Construction, Operation and Testing) and CORAL (block structured language, based on ALGOL60). Their test environment consists of the following tools, called TAS (Test Aid Suite):

- *Unit Driver* – A package which is independent of the software under test, that provides test harness and allows initialisation of set up values, specification of a unit (or a part of a unit) to be executed, and comparison of results obtained to those expected.
- *Path analyzer* – Partitions the source code into SubPath Modules (SPM's). An SPM is a basic block containing no branch points. A sequence of SPMs forms a subpath.
- *Instrumenter* – Adds to the source code necessary "calls", to provide "execution history", debug facilities, test coverage analysis, and static analysis concerning the source structure.

Conceptually, this approach is very similar to SADAT (although testing a structured language) and suffers the same disadvantages.

4.2.2 Statistical Testing

The temptation to use a "sampled" test set, originates in the fact that the amount of different inputs required to test a program systematically may become enormous [30]. Yet, the smaller the sample is – the lower the reliability is, therefore a decision upon the sample size must be calculated carefully. Typical results of statistical testing methods are: an expected value, a risk, a probability, confidence limits / levels of significance, variances. In real time software testing a particular emphasis is put on:

- Deadlock occurance.
- Correctness of a sequence of outputs.
- Occurance of final/intermediate results in the right time interval.

Risk Calculation

The statistical testing tries to save test runs, and thereby to reduce cost, but an unfortunate fact of decreasing the 100% proof of the system is introduced. A comparison between the cost reduction and the risk involved is therefore necessary. One way of defining the *risk cost* is [30]:

$$r = \sum_{a} H(a)X(a)$$

, where:

- $X(a)$ is the lost caused by event a.
- $H(a)$ is the frequency of "loss causing event" a, during a relevant period of time.
- $H(a) = \sum_i H(i)P(a|i)P(a|a_i)$.
- $H(i)$ is the frequency of "initiating event" i which may cause a.
- $P(a|i)$ is the probability that the system under test will fail to react correctly on i.
- $P(a|a_i)$ is the probability that any alternative action (installed previously) fails simultaneously.

Simple Cases

An analysis of the probability of detecting error-occurances [30] is sketched below.

First, examining the probabilities of "hitting" an error.

- The probability of hitting one error, associated with time interval D, in the program run time T, with no condition is: $P_{w_1} = D/T$. Hence, for N test runs: $P_{w_N} = 1 - (1 - D/T)^N$.
- The probability of hitting one error in one test run, now with one binary condition, is: $P_{b1} = 1/2D/T$. For N test runs and k binary conditions: $P_{bN} = 1 - (1 - (1/2)^k \times D/T)^N$.

Now, consider the case of sequence of tasks. When M tasks access one resource, there are $M!$ possible access sequences. If we assume equal probability of failure for all accesses, then the probability of detecting one failing access, in n runs, is

$$P = 1 - (1 - 1/M!)^n$$

. If the M tasks access N resources, the number of possible access sequences becomes:

$$K = \sum_{j=1}^{N} \sum_{i=1}^{M} (n_i \times m_{i,j})$$

.

- n_i the number of runs of task i.
- $m_{i,j}$ the number of accesses to resource j during one run of task i.

If all possible accesses are to be considered, then the number of sequences becomes $K!$, while the probability of hitting error is known from above.

The presence of several possible priority interrupts in a time interval may trigger several task sequences, or require queue rearrangements. These cases are combinatorially treated the same as the access problem above, keeping in mind that deadlock problem may occur in a specific sequencing.

Testing Large Systems

For large systems a slightly different approach is suggested in [30]. Defining p_0 to be the probability of one distinct arbitrary property, we would like to obtain the number of properties (denoted N_t) we have to test for, in order to ensure $p_0 < P_0$, within a confidence level $CL\%$. Assuming binomial distribution of properties' error detection probability, the number of properties we have to test for: .

$$N_t = 4.6/P_0 \text{ for } CL = 99\%$$
$$N_t = 3.0/P_0 \text{ for } CL = 95\%$$

If P_0 is between 10^{-7} to 10^{-4}, N_t becomes enormous. Reducing it is possible only by relating to previous analysis, performed analytically and not empirically. Obtaining constraints yields simplification of the testing problem.

Problems With Large Tests

The large number of runs required for a reasonable confidence level sets additional requirements to those found in the simple cases:

- Testing real time properties is implementation (and hardware) dependent by nature. Hence, the system should be tested as a whole, while the tested computer is activated by a computerised test equipment (TE), since we require a large number of runs.
- The TE provides the test inputs, controls the timing, and monitors the outputs. Hence the reliability required from the TE is very high.
- If a random number generator is included in the TE, its repetition period should be sufficiently long.

- The statistical analysis shows clearly that in cases where we have to maintain reasonable confidence levels, reduction of test sequences is achieved mostly due to formal analysis of the system under test, rather than by the statistical approach itself.

4.3 Analysis and Proof

All testing methods we reviewed have shown clearly that in order to obtain meaningful test set-ups, a great deal of effort should be invested in analysing and formally proving properties of the tested program. The earlier this effort is invested during the development cycle the more benefits can be gained: by better phrasing of the problem, ability to predict the solution's behavior, and pinpointing bottle-necks and weak points. If such attitude is adopted, proof systems should give answers to all system's phases, from early model to the detailed source code, proving correctness of solution strategy as well as correctness of the implementation. Since the main interest of this work is real time programs, the main part of this section will concentrate on systems that try to verify and prove timing properties.

4.3.1 Process Based Model Analysis

Categories

There are two major approaches in analysing and demonstrating program's properties without execution, assuming a process based model has been prefered:

- Proving that the program satisfies certain criteria, or performs according to given specifications.
- Proving by analysis certain structural properties of the model.

The following example will demonstrate these kinds of analysis.

Example: Flow-graphs

Two bi-digraphs are used to model the system [1]. A *control flow-graph* describes the structural behavior of the program, and the control flow during execution, while a *data flow-graph* (corresponding to each execution sequence) describes the data behavior during this execution [22]. In the control graph: the *vertices* represent *control points*, and the directed *arcs* represent *actions* or control transitions. In the data flow-graph: there are two types of *vertices: data items* and *operations*. The vertices are connected by directed *arcs* describing the data *flow*. A data flow-graph corresponds to an execution sequence $S = (a_1, .., a_n)$

[1] Modeling a system in a flow-graph method is described in section 2.4.2 of this document. A brief review is repeated here.

in the control flow-graph, by attaching to each arc a_i a mapping of input variables $(X_1, .., X_k)$ into output variables $(Y_1, .., Y_m)$. This functional relation is the vertex of type "operation" which appears in the graph. The graph, which may be very large in case of a complex program, may be reduced by means of abstraction levels, merging data items to vectors, and sequential actions into control segments. This graph may be used to demonstrate structural properties and to verify some performance ones.

- Independent data items may be detected, and point the distributed implementation that requires less communication traffic. The problem becomes a partitioning problem which requires that the number of arc-cuts is minimised.
- Each execution sequence with its corresponding data flow-graph encounters all the information needed for numerical error-bound analysis.

Deficiencies

Flow graphs and more advanced tools, such as SPECK [30], share the same deficiencies. Program correctness verification is very difficult to implement, and more difficult to understand. Timing properties are either ignored at all or receive a simple and fruitless analysis. The statistical nature of inputs is not considered, and average and peak performance evaluation is not provided.

4.3.2 Finite State Automata Model Analysis

Graph Model Analysis

The usage of graph models to describe problems and solutions has spread during the last decade. The graph models were described in chapter 2 of this document, and this section will concentrate in their analysis power. This modeling provides information about the structure of the solution, as the process based method does. Yet, more properties can be analysed: safeness, boundedness, liveness (deadlock freedom), reachability of states, equivalence between solutions (optimisation), as well as timing properties.

Various graph methods have been developed and used. Some examples are the bipolar synchronisation graph, the R-nets in SREM [4], and the Petri nets. Some of them share a lot in common, and some look as an augmentation of others. The next pages will describe a combined use of stochastic properties and the Petri nets.

Stochastic Petri Network (SPN) Model Analysis

Classic Petri nets [31] do not contain any timing properties concerning the behavior of the finite state machine they describe. The *reachability set* which is produced when analysing the net, describes all possible states the machine

can reach. If this model is extended [27], by assigning *probabilistic firing-rates* to transitions, the net becomes isomorphic to an homogeneous Markov process, due to the countability of the markings and the memoryless property of the firing rate. Some definitions summarise the SPN.

- $PN = (P, T, A, M)$, A Petri network.
- $SPN = (P, T, A, M, Q)$, PN extended to SPN.
- $P = \{p_1, .., p_n\}$, Places - drawn as circles.
- $T = \{t_1, .., t_m\}$, Transition - drawn as bars.
- $A = \{P \times T\} \cup \{T \times P\}$, Input and output arcs.
- $Q = \{q_1, .., q_m\}$, Average transition rates, for the exponentially distributed firing times.
- $M = \{m_1, .., m_n\}$, Initial marking - drawn as dots. $M : P \rightarrow \mathcal{N}$ (the natural numbers), $M(p_I) = m_I I = 1, .., n$.
- Set function $I(t) = \{p : (p, t) \in A\}$, Input places for transition t.
- Set function $O(t) = \{p : (t, p) \in A\}$, Output places for transition t.

Molloy ([27]) proves that any finite-place, finite-transition, marked SPN is isomorphic to an one-dimensional discrete-space Markov process, concerning the marking sequence. Hence, in addition to all properties that can be proved using a regular Petri net (as describing concurrency, contention and synchronisation, analysis of deadlocks boundedness and self regulation), the SPN provides capabilities of performance verification: analysis of average delay, average throughput etc., using queueing theory [23]. Consider the following simple example (see Figure 4.1) which demonstrates the combined use of the above methods.

Example: Given the following SPN:

1. The *places* and *transitions* sets:
   - $T = \{t_1, .., t_5\}$.
   - $P = \{p_1, .., p_5\}$.
2. The connections:
   - $I(t_1) = \{p_1\}, O(t_1) = \{p_2, p_3\}$.
   - $I(t_2) = \{p_2\}, O(t_2) = \{p_4\}$.
   - $I(t_3) = \{p_3\}, O(t_3) = \{p_5\}$.
   - $I(t_4) = \{p_4\}, O(t_4) = \{p_2\}$.

- $I(t_5) = \{p_4, p_5\}, O(t_5) = \{p_1\}$.

3. The firing probabilities: $q_1 = 2, q_2 = 1, q_3 = 1, q_4 = 3, q_5 = 2$.

4. The initial marking: $M = \{m_1 = 1; m_I = 0, I \in 2..5\}$.

Solving the above system using Molloy's method is performed as follows.

1. Analysing the net structure, the transitions can be characterized as:

   - (t_5, t_1) – Sequential.
   - (t_2, t_3) – Parallel.
   - (t_4, t_5) – Contention.
   - t_1 – Fork. t_5 – Joint.

2. The reachability set, describing the "token" occupancy in the places set (i.e. the marking) $M_i = (p_1, p_2, p_3, p_4, p_5)$:

   - $M_1 = (1, 0, 0, 0, 0)$.
   - $M_2 = (0, 1, 1, 0, 0)$.
   - $M_3 = (0, 0, 1, 1, 0)$.
   - $M_4 = (0, 1, 0, 0, 1)$.
   - $M_5 = (0, 0, 0, 1, 1)$.

   A set of five reachable marking states.

3. Solving the ergodic Markov chain (as in [23]):

   - $2Pr[M1] = 2Pr[M5]$.
   - $2Pr[M2] = 2Pr[M1] + 3Pr[M3]$.
   - $4Pr[M3] = Pr[M2]$.
   - $Pr[M4] = 3Pr[M5] + Pr[M2]$.
   - $5Pr[M5] = Pr[M4] + Pr[M3]$.
   - $Pr[M1] + Pr[M2] + Pr[M3] + Pr[M4] + Pr[M5] = 1$.

4. The marking steady state probabilities:

   - $Pr[M1] = 0.1163$.
   - $Pr[M2] = 0.1860$.
   - $Pr[M3] = 0.0465$.
   - $Pr[M4] = 0.5349$.
   - $Pr[M5] = 0.1163$.

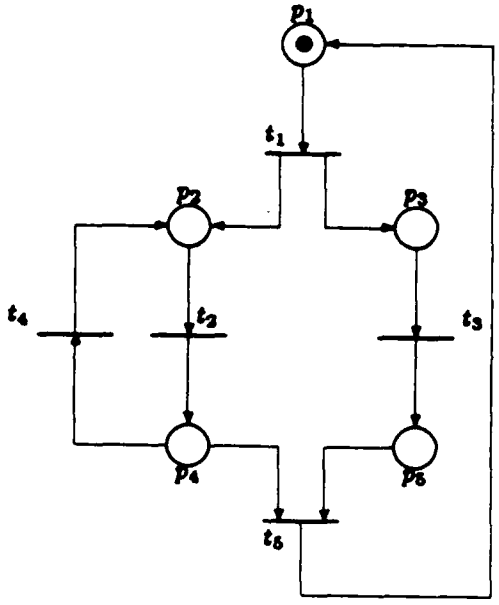


Figure 4.1: Example.

5. Calculating the probabilities of token occupancy:

   - $Pr[m_1 = 1] = 0.1163 = Pr[M1]$.
   - $Pr[m_2 = 1] = 0.7209 = Pr[M2] + Pr[M4]$.
   - $Pr[m_3 = 1] = 0.2325 = Pr[M2] + Pr[M3]$.
   - $Pr[m_4 = 1] = 0.1628 = Pr[M3] + Pr[M5]$.
   - $Pr[m_5 = 1] = 0.6512 = Pr[M4] + Pr[M5]$.

6. The probabilities calculated above depend on the reachability set, which itself depends on the initial marking. If the initial marking was 2 tokens in p_1, the reachability set would consist of 14 states, and 3 tokens in p_1 would produce 30 states.

7. The ergodicity of the chain allows using flow balance technique and Little's law for average performance analysis. In this example, the *average* delay time can be calculated as follows. Transition t_1 can be enabled only if p_1 contains a token, hence utility(t_1)=0.1163. Having $q_1 = 2$, implies an average token flow of 0.2326 tokens per time unit in p_1. Since t_1 is a fork, the average token flow in the parallel paths is doubled (0.4652), and reduced again in the joint t_8. Since the system conserves tokens (neither destroy nor produce), we can apply Little's law, knowing the average flow rate in each branch from above,

   - $T_{av} = N_{av}/q_{av}$.
   - $N_{av} = \sum_{l=1}^{5} m_{lav} = 1.7674$.
   - $T_{av} = 1.7674/0.4652 = 3.8$ time units.

Another approach (Generalised Stochastic Petri Nets) is introduced in [3]. Combining this approach with *product form queueing network* theory provides a tool for performance verification for non-classical queueing constructs [7]. This combination can also be used to find upper and lower bounds, as an approximation for non-product-form systems performance.

4.3.3 Theorem Proving Techniques

Proving in General

Hoare's axiomatic approach was adopted for proving programs in various proof systems. This approach was extended for distributed systems [29], and constitutes a base line for other proof systems, as for CSP proving [6] and others. There are some major difficulties using the axiomatic approach:

- Time dependent properties of concurrent systems (concurrency, mutual exclusion) are difficult to specify.

- Finding invariants for complex systems.
- Simplification of long expressions are in most times tedious.

The following paragraph is dealing with an approach which tries to solve these problems.

Event Based Model

Event based model of a system [2] ([9]), separates the systems's properties into two major categories: *behavior* which mainly concerns the external view of the system, and *structure* which reflects the internal view of the system. Proving the correctness of a system is translated into a consistancy check between the behavior and the structure. Orthogonality between properties allows verification of each independently, and thereby avoids "exponential state explosion" when coming to derive test sets. The model is constructed from events and their relations:

1. An *event* is an *instantaneous* (takes zero time) *atomic* (happens completely or not at all) state transition in the system's computation history.
2. *Time ordering* is achieved with the *precede* ($\rightarrow$) relation, which is a partial ordering found also in [24]. Event $e1$ precedes event $e2$ ($e1 \rightarrow e2$) if:
   - $e1, e2$ are events at the same process (an autonomous computation node, having its own local clock) and $e1$ comes before $e2$, or
   - $e1, e2$ are events at different processes, and $e1$ is a *send message* event, and $e2$ is a *receive* event of the very same message.

   This partial ordering ensures that $e1 \rightarrow e2$ implies that $e1$ happens before $e2$ by any measure of time. (It is not IFF !). The ordering is transitive, irreflexive and anti-symmetric.
3. *Causality ordering* is achieved with the *enable* ($\Longrightarrow$) relation, which is also a partial ordering, and is defined as follows:

   Event $e1$ enables event $e2$ ($e1 \Longrightarrow e2$) IFF the *existance* of event $e1$ will *cause occurance* of even $e2$ in the future.

   This ordering is also transitive, irreflexive and anti-symmetric.

This model allows program verification in a theorem proving fashion, using the model's definitions combined with first order predicate calculus. Orthogonality of relations is used to simplify proof procedures which are complex, yet abstraction levels must be enforced to deal with very large systems. The system interacts with its environment by exchanging messages through *unidirectional ports*, in which local history is ordered locally, and the *behavior* properties are specified and measured. Some good property verifications are:

[2] The event-based model is described in section 2.5, and a brief description is repeated here.

- *Concurrency* of events $e1$ and $e2$ is proved IFF $\neg(e1 \rightarrow e2)$ and $\neg(e2 \rightarrow e1)$.
- *Liveness* of event is proved applying an "enable"s sequence to the initial state. It guarantees that event will *eventually* happen, (proving starvation freedom or message delivery) but more strict timing requirements cannot be proved by this proof system which provides only partial-order relations.

Real Time Logic

Another event based model is introduced in [21], in order to verify safety properties of a real time system. The model consists of events, actions, causality relations, and timing constraints. The model is expressed in a first order logic, describing the system properties as well as the system's dependency on external events. The Real Time Logic system (RTL) captures time with an event occurance function denoted "@". This function assigns time values to event occurances, while the constraints and the scheduling disciplines are restrictions imposed on the function. RTL uses three types of constants:

1. *Action constants* – may be primitive or composite. In a composite constant precedence is imposed by the event-action model using sequential or parallel relations between actions.
2. *Event constants* – are divided to three classes. Start/stop events describe the initiation/termination of an action or subaction. Transition events are those which make a change in state attributes (i.e. a change in an assertion about the state of the real time system or its environment). External events are those which cannot be caused by the system, but can impact system behavior.
3. *Integers* – assigned by the occurance function, to capture time, and used to denote the number of an event occurance in a sequence.

Assertions about the physical state of the system over time are translated into *algebraic relations*, involving the occurances of the appropriate transition events. *State predicates* are used as a notational device for asserting truth-values to state attributes during a time interval.

A set of axioms can be derived from the event-action model of the system by applying an automatic translation to the system specification. This translation describes the relations between actions and their start/stop events, the sporadic and periodic events constraints, as well as the causal relations which may initiate a transition event. Artificial constraints may be added in order to prevent the scheduler from executing actions that are not counted towards meeting specified timing constraints (i.e. not required), especially when utilisation of resources is less than 100%.

A timing property of a system (an RTL assertion) is expressed by showing that there is no occurance function which is consistent with the system specification, in conjunction with the complement of this particular property. The

mechanism to achieve it is the *deductive resolution*. An important characteristic of RTL allows using procedures used in Presburger Arithmetic: an RTL formula consists of only algebraic relations and state predicates connected by first order logic operators.

An advantage of RTL is the uniform way in which different types of constraints can be expressed, by means of algebraic relations of event occurances. Yet, it is weak in providing hierarchy levels of abstraction, which are necessary to simplify system examination, and to relate the implemented solution to the requirements specification. Since this approach is very young, it will probably be developed in the future to provide these features.

4.3.4 Timing Properties analysis

Weakest Pre-condition and Predicate Transformers

A very interesting approach is introduced by V. Haase [17] for verification of real time behavior of programs. Three major assumptions are the base of this approach:

1. The program consists of parts which are sequential and parts which are parallel. The sequential parts are constructed from Dijkstra's *guarded commands*, and the parallel constructs (*PARCs*) are CSP-like parallel interpretation of the guarded commands.

2. The weakest *precondition* predicate, "wp", provides the execution time properties to the program parts:

   - In case of a *simple statement (S)*, i.e., not an iterative nor a conditional one - whose execution time can be defined as the non-interrupted execution time needed for the implementing *processor* - the weakest precondition can be interpreted as "the latest starting time" t to meet deadline T with statement execution time t_s:

     $$wp(S, t \leq T) = t \leq T - t_s$$

   - In case of an action, i.e., a non-interrupted sequence of sequential statements ($action = S_1; ..; S_n$),

     $$\begin{aligned} wp(action, t \leq T) &= wp(S_1, wp(S_2, ..wp(S_n, R)..)) \\ &= t \leq T - \sum_{i=1}^{n} t_{si} \\ &= t \leq T - t_{action} \end{aligned}$$

3. Since execution is also *input-data-dependent*, and not only hardware dependent, this property (which appears in branching points and in iteration decision) is characterized with Dijkstra's *predicate transformer rule*

   $$t_{action} = f(d_1, .., d_n)$$

where $d_1, .., d_s$ are input data of *action*.

In practice one can distinguish two cases:

- when *f(..)* is a constant or very *simple*, then execution time can be evaluated *explicitly* prior to execution.
- when *f(..)* is comparatively *complex*, then an *upper bound* can be estimated, and if the bound is passed during execution, a special process (as "watch-dog") may re-evaluate necessary updates.

Time Behavior of Sequential Programs

Haase suggests a method to analyse sequential programs:

1. *Transformation into guarded-command notation:* Transforming the non-iterative and non-branching parts is done as described above, while deriving the weakest-preconditions appropriately. Then
   - *Conditional statements* are transformed into Dijkstra's IF:

     IF
     $g_1 \rightarrow action_1$
     .
     .
     $g_n \rightarrow action_n$
     FI

     Deriving the weakest precondition:
     $wp(IF, t \leq T) = (\exists j : g_j)$ and $(\forall i : g_i \rightarrow t \leq T - t_{action_i})$.
   - *Iterative statements* are transformed into Dijkstra's DO:

     DO
     $g_1 \rightarrow action_1$
     .
     .
     $g_n \rightarrow action_n$
     OD

     Deriving the weakest precondition:
     $wp(DO, t \leq T) = (\exists k \geq 0 : H_k(t \leq T))$, where
     $H_0(t \leq T) = (t \leq T)$ AND $\neg(\exists j : g_j)$, and
     $H_k(t \leq T) = wp(IF^+, H_{k-1}(t \leq T))$ OR $H_0(t \leq T)$ [3] .

[3] IF^+ denotes the same guarded-command set with assumed IF/FI brackets.

2. After the program is written in an equivalent guarded commands notation, the *programs weakest precondition* has to be *constructed*, applying the sequential rules introduced above.

3. *Evaluate* the program's *wp*: The evaluation of the program's weakest precondition

$$wp(program, t \leq T) = wp(S_1, ..wp(DO, wp(IF, t \leq T))..)$$

is carried out "inside first". Starting with the inner most *wp*, its weakest precondition is evaluated, then substituted in the following, until the outer most is evaluated, and thereby provides the whole program's latest starting time to meet the required deadline.

Time Behavior of Parallel Programs

Parallel programs, constructed with PARCs, can be analysed with predicate transformers as well. A very important assumption is taken in the constructing phase, that the *guards* variables are *mutually exclusive*. This yields a very similar guarded commands set, yet the *real time behavior is different* from the sequential programs.

- *Parallel condition construct*, denoted IF-PARC, is carried out as follows:

PAR-IF
$$\begin{array}{l} g_1 \rightarrow action_1 \\ \| \ . \\ \| \ . \\ \| \ g_n \rightarrow action_n \end{array}$$
PAR-FI

Semantics are defined by the formulas

$$(R_1) and (R_2) .. and (R_n) \rightarrow R$$

and

$$wp(IF - PARC, R_1 .. and R_n) =$$

$$(\forall i \in 1..n) : (g_i \rightarrow wp(action_i, R_i) and \neg(g_i) = R_i) \rightarrow wp(IF - PARC, R).$$

All R_i are substituted by $t \leq T$, and false g_is are omitted (not contributing to execution time). The latest starting point is therefore

$$wp(IF - PARC, t \leq T) = \forall i : g_i \rightarrow (t \leq T - t_{action_i}).$$

- *Parallel iterative construct*, denoted DO-PARC, is carried out as follows:

PAR-DO
$g_1 \rightarrow action_1$
||.
||.
|| $g_n \rightarrow action_n$
PAR-OD

Semantics identity between DO and DO-PARC allows using parallel structures as well:

$$wp(DO - PARC, t \leq T) = (\exists k \geq 0 : H_k(t \leq T)).$$

But the *evaluation* is *different*:

$$\text{DO: execution time} = t_{ai} + t_{aj}$$
$$\text{DO-PARC: execution time} = max(t_{ai}, t_{aj}).$$

A method to evaluate the latest starting point is

1. Each *action* is represented by a *vector* in *the state space* of PARCs.
2. Establish the *paths*, i.e. the sequences of *actions*, leading from assumed *pre-states* to given *post-states* of the whole activity.
3. Consider the following cases:

   Case a. If the path between any two states is *unambiguous*, then the *sum* of the intermediate execution times of the steps is taken into account.

   Case b. If the path is *ambiguous*, then *determine* partial sequences that can be *exchanged*, and then *maximum* execution time of the partial sequences is taken into account.

4. If in the example above we assign equal execution times to the actions of P_i (every vertical step), denoted by t_v, and equal execution times to the actions of P_j (every horisontal step), denoted t_h, we can conclude (see Figure 12):

   4 to 3:$action_j \rightarrow t_h$
   3 to 1:$action_j || action_i \rightarrow max(t_h, t_v)$
   1 to 0:$action_i \rightarrow t_v$

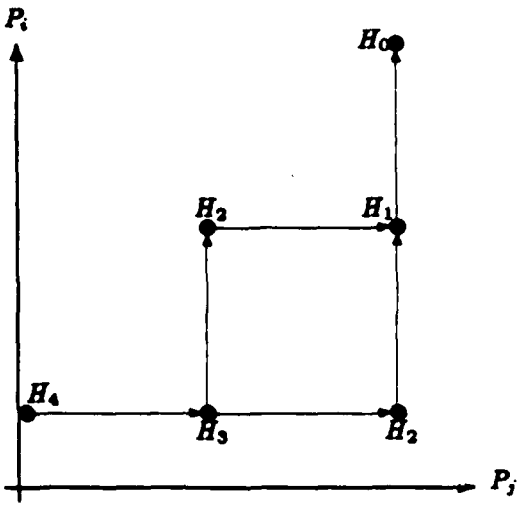


Figure 4.2: Paths Example.

Therefore, the calculation of the weakest precondition is restricted by

$$t \leq T - t_h - t_v - max(t_h, t_v).$$

Haase's approach gives a tool to deal with medium-complex real-time programs, yet when dealing with very complex systems, one may find it very hard to analyse. In order to solve this situation, he suggests the use of a *dynamic checking of* (estimated) *deadline*, by adding a parallel *control action*. This dynamic scheduling approach is very similar to the *Latency Scheduling* we reviewed in chapter 2.

4.3.5 Operational Analysis

Performance evaluation using [12] operational analysis provides a quantifying tool to obtain average performance properties. Using a *routing connection* between *service centers* (C.S.) whose performances are given, one can obtain the performance of a specific *system configuration* in steady state. It is done assuming *job flow balance*, and applying simple statistical tools to derive properties which one cannot obtain by observation (external direct measurement). Although these properties may be derived by the models we reviewed so far, it seems that this evaluation technique covers a field which all the other papers (mentioned in this review) have not dealt with. This is the *resources saturation*

problem, which arises due to the resources *bounded utilization*, which produces the *bottle-neck* phenomena. Analysing the system's performance must take into account the simple fact that service centers may be "pushed" only up to their utilisation upper bound, and therefore, the weakest-point in the system configuration will slow down all other parts.

In order to describe the *bottle-neck analysis*, the notation used in [12] will be used in the following paragraph. Let

- U_i be the *utilization* of S.C. i, defined as its "busy time" over an observation period T.
- X_i be the *throughput* of S.C. i, defined as the number of job "departures" from it over an observation period T.
- V_i be the *visit ratio* of S.C. i, defined as its throughput part in the whole system throughput (equals X_i/X_o).
- S_i be the *mean service time* of S.C. i, defined as the portion of its busy time served to each customer (i.e. $(T \times U_i)/(T \times X_i)$). Hence, $U_i = X_i \times S_i$
- R_i be the *mean response time* of S.C. i, and let R_o denote the whole system response time for a single customer.

The bottle-neck analysis is done assuming a device has reached its highest utilisation, and let this device be denoted by the subscript b. Hence, $U_b = 1$. In order to determine which S.C. is b, we have to compare the utilisations of all S.C.s, and pick the highest one, since increasing the customers number will increase utilisations of all S.C.s, until one of them will reach *1*, and the first to do so will be the one with the highest U_i to begin with.

$U_i/U_j = (X_i \times S_i)/(X_j \times S_j)$
division and multiplication by X_o gives
$U_i/U_j = (X_i \times S_i/X_o)/(X_j \times S_j/X_o)$
and since $X_i/X_o = V_i$ then
$U_i/U_j = (V_i \times S_i)/(V_j \times S_j)$
Then, in order to pick the highest utility,
$V_b \times S_b = MAX_i\ (V_i \times S_i)$

For saturation state of the bottle-neck S.C. the following will hold:

$U_b = 1 \Longrightarrow X_b = 1/S_b$
$V_b = X_b/X_o$
Hence,
$X_o = 1/(V_b \times S_b)$

Due to $N = N_*$ customers in the system.

Yet, for the single customer case, $X_o = 1/R_o$ due to $N = 1$ customer in the system.

Plotting a graph of the system throughput versus the number of customers in the system, we have the two points (for $N = 1$ and $N = N_*$) with their corresponding throughput, to construct the asymptotes which serve as an upper bound to the systems throughput. The saturating N can be calculated as

$N_*/1 = (1/V_b \times S_b)/(1/R_o)$
for a system without delay.
Hence,
$N_* = R_o/(V_b \times S_b)$.

The results of the latter analysis are based on broad assumptions: mean service time for all customers, similar demands of service within time, etc. No peak analysis can be derived from them, yet they form a performance bound that has to be considered when the system is evaluated.

4.4 Simulation As Verification Tool

4.4.1 Classes and Aims

One can distinguish two major classes of simulations that are used to verify properties of a real time program:

1. Simulation of the system under test itself.
2. Simulation of a plant/load that the system meets as its "real world".

There are various reasons to *simulate the program itself*:

- During the first phases of the design, it helps to verify basic properties of the model used, i.e. the approach chosen to solve the given problem.
- Using an approximate simulation, one may predict approximate performance of the system, within a certain confidence level.
- Design trade-offs may be checked and analyzed.
- After the system completes its development cycle, a detailed simulator may be used to verify achievement of design goals, by comparing its outputs to the real system's outputs.
- It may serve as a good tool for providing answers to "what if" questions, especially when deciding on upgrading the system.

A detailed simulator of a system is very expensive to develop [30], and the effort invested in it may lead to the fact that it is completed only after the real system is ready [5]. A more common simulation is a *plant/load simulator*, which is used for:

- Controlled measurements of the system under load, making it possible to isolate specific properties in a specific environment.
- Proving system under test, when there is a danger to test it with the real plant (e.g. control program of a nuclear power plant, control program of a weapon), if the risk calculations (see section 4.2.2) justify the effort of developing such a tool.
- A good debugging tool when trying to reconstruct a pattern that lead to a crash or a deadlock, when only partial information exists for the analysis.

The basic idea in using simulation as a verification tool, is to gain an advantage that no test provides. A test is constructed according to the system's specification. The simulation provides a *verification of the specification* as well as of the system under test.

4.4.2 Problems in Simulation As a Verification Tool

1. Simulation of a large system is rarely used, especially due to the cost involved in developing it. A good simulator of a system, requires an effort investment which is in the same order of magnitude as the development of the system itself.
2. When the risks of operating the system are high, the simulator used as a test tool should be highly reliable, to an extent even more than the system under test. This means that in addition to the development cost, the simulator has to be extensively used per se, before being qualified as a verifying tool.
3. If the design model and the simulator are derived from the same basic assumptions, a special kind of errors may arise, called *common mode errors*: The simulator and the system under test are both mistaken, and the errors are failed to be detected.
4. When comparing simulator results and system results, discrepancies may be found. The problem of deciding "Who is wrong?" may be very difficult to solve. One must be very cautious not trying to change the real world.
5. The most difficult problem in simulating a large real time system is the dependency of its performance on the sequence of inputs data. Since the simulator is required to perform as the real system should, including real time properties, the complexity required from it is of the same level, and sometimes even higher.

The problems described above make simulation as a verification tool rarely found when dealing with large real time systems. Only in cases where it is unavoidable, regarding risk factors involved, a large effort is invested in it.

Chapter 5

Conclusion

This work tries to review three phases in the life cycle of real time programming. Chapter 2 of this document reviews the design phase, by introducing requirements specification techniques, modeling methods, and basic approaches to the design as a whole. Chapter 3 of this document reviews implementation aspects of real time programming, by introducing a disciplinary approach, as well as focusing on implementation aspects that concern the Ada language tasking facilities. Chapter 4 of this document reviews the problematic aspects of validating a real time system. Various methods of testing are presented, as well as proof systems and simulation techniques.

As can be seen in the above chapters of this document, methods that are powerful in one phase, seem to fail in another. For example, a stochastic approach of any type provides powerful results in verifying mean values of a system, but fails to serve as an aid for overcoming scheduling problems in the design phase. This situation sometimes leads to a combined use of some methods, linked together by some means, to provide a scheme which covers all the aspects (as in [5]). In this review, one may find such links. One of them is the Petri net, which can be used in the design ([11]) with some augmentations ([10]), it can be automatically translated to a program, with some modifications and annotations ([28]), and it can also be used to verify some statistical properties ([27,3,7]). Operational approach also tries to bridge the different phases ([35]), as do the structured methods ([4,22,16]). Some theoretical considerations ([26]), and disciplinary recommendations ([34,18]) may assure that the combined use of methods is well coordinated.

As stated in the beginning of this paper, the objective here has been to describe the major techniques applicable to the three phases in the life cycle of a real-time system. In this regard, we attempted to include most major techniques and trends for the real-time systems.

Bibliography

[1] Ackscyn R., McCracken D., *Zog and The USS Carl Vinson: Lessons in System Development*, CMSC Dept. Carnegie Mellon Univ., Pittsburgh, PA, March 1984.

[2] *Reference Manual for The Ada Programming Language*, U.S. DOD (ANSI) MIL-STD 1815a-1983, Feb 1983.

[3] Ajmone-Marsan M., Balbo G., Conte G., *A Class of Generalized Stochastic Petri Nets for Performance Evaluation of Multiprocessor Systems*, ACM Trans on Computer Systems, Vol 2 No 2 pp 93-122, May 1984.

[4] Alford M., *A Requirement Engineering Methodology for Real Time Processing Requirements*, IEEE Trans on Software Engineering, Vol SE-3 No 1 pp 60-69, Jan 1977.

[5] Anderson G., *The Coordinated Use of Five Performance Evaluation Methodologies*, Communications of the ACM, Vol 27 No 2 pp 119-125, Feb 1984.

[6] Apt K., Frances N., De Roever W., *A Proof System for Communicating Sequential Processes*, ACM Trans on Programming Languages and Systems, Vol 2 No 3 pp 359-385, July 1980.

[7] Balbo G., Bruell S., Ghanta S., *Combining Queueing Network and Generalized Stochastic Petri Nets Models for the Analysis of Some Software Blocking Phenomena*, IEEE Trans on Software Engineering, Vol SE-12 No 4 pp 561-576, April 1986.

[8] Basili V., Weiss D., *Evaluation of Software Development by Analysis of Changes*, TR-1236 Univ. of MD, Dec 1982.

[9] Chen B., Yeh R., *Formal Specification and Verification of Distributed Systems*, IEEE Trans on Software Engineering, Vol SE-9 No 6 pp 710-722, Nov 1983.

[10] Coolahan J., Roussopoulos N., *Timing Requirements for Time Driven Systems Using Augmented Petri Nets*, IEEE Trans on Software Engineering, Vol SE-9 No 5 pp 603-616, Sept 1983.

[11] Courvoisier M. et al, *Task Synchronization in Distributed Real Time Control Systems*, IEEE Proceedings - Real Time Systems Symposium, pp 83-88, Miami Beach FA, Dec 1981.

[12] Denning P., Buzen J., *The Operational Analysis of Queueing Network Models*, Computing Surveys, Vol 10 No 3 pp 225-261, Sept 1978.

[13] Dowling J., *Some Methods and Tools for Real Time Software Validation*, Proceedings of The 12'th IFAC/iFIP Workshop of Real Time Programming, pp 81-86, Hatfield UK, 29-31 March 1983.

[14] Garetti P., Laface P., Rivoira S., *Multiprocessor Implementation of Tasking Facilities in Ada*, Proceedings of The 12'th IFAC/IFIP Workshop of Real Time Programming, pp 97-102, Hatfield UK, 29-31 March 1983.

[15] Gehani N., Cargill T., *Concurrent Programming in Ada Language: The Polling Bias*, Software Practice and Experience, Vol 14 No 5 pp 413-427, May 1984.

[16] Gomma H., *Software Design Method for Real Time Systems*, Communications of the ACM, Vol 27 No 9 pp 938-949, Sept 1984.

[17] Haase V., *Real Time Behavior of Programs*, IEEE Trans on Software Engineering, Vol SE-7 No 5 pp 494-501, Sept 1981.

[18] Heninger K., *Specifying Software Requirements for Complex Systems: Techniques and Applications*, IEEE Trans on Software Engineering, Vol SE-6 No 1 pp 2-13, Jan 1980.

[19] Hoare C., *Communicating Sequential Processes*, Communications of the ACM, Vol 21 No 8 pp 666-677, Aug 1978.

[20] Houghton , *Software Development Tools*, National Bureau of Standards, Special Publication 500-74, 1982.

[21] Jahanian F., Mok A., *Safety Analysis of Timing Properties in Real Time Systems*, Department of Computer Science, University of Texas, Austin, Texas, Sept 15 1985. (To appear in IEEE Trans on Software Engineering).

[22] Kodres U., *Analysis of Real Time Systems by Data Flowgraphs*, IEEE Trans on Software Engineering, Vol SE-4 No 3 pp 169-178, May 1978.

[23] Kleinrock L., *Queueing Systems*, John Wiley and Sons, New York NY, 1975.

[24] Lamport L., *Time, Clocks and Ordering of Events in a Distributed System*, Communications of the ACM, Vol 21 No 7 pp 558-565, July 1978.

[25] Ma P., Lee E., Tsuchiya M., *Design of Task Allocation Scheme for Time Critical Applications*, IEEE Proceedings - Real Time Systems Symposium, Miami Beach FA, Dec 1981.

[26] Mok A., *Fundamental Design Problems for the Hard Real Time Environment*, MIT Ph.D. Dissertation, Cambridge MA, May 1983.

[27] Molloy M., *Performance Analysis Using Stochastic Petri Nets*, IEEE Trans on Computers, Vol C-31 No 9 pp 913-917, Sept 1982.

[28] Nelson R., Haibt L., Sheridan P., *Casting Petri Nets into Programs*, IEEE Trans on Software Engineering, Vol SE-9 No 5 pp 590-602, Sept 1983.

[29] Owicki S., Gries D., *Verifying Properties of Parallel Programs: An Axiomatic Approach*, Communications of the ACM, Vol 19 No 5 pp 279-285, May 1976.

[30] Quirk W. (editor), *Verification and Validation of Real Time Software*, Springer-Verlag, Berlin Germany, 1985.

[31] Reisig W., *Petri Nets*, Springer-Verlag, Berlin Germany, 1985.

[32] Teichroew D., Hershey E., *PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems*, IEEE Trans on Software Engineering, Vol SE-3 No 1 pp 41-48, Jan 1977.

[33] Voges U., et al, *SADAT - An Automated Testing Tool*, IEEE Trans on Software Engineering, Vol SE-6 No 3 pp 286-290, May 1980.

[34] Wirth N., *Toward a Discipline of Real Time Programming*, Communications of the ACM, Vol 20 No 8 pp 577-583, Aug 1977.

[35] Zave P., *The Operational Versus The Conventional Approach to Software Development*, Communications of the ACM, Vol 27 No 2 pp 104-118, Feb 1984.

END

7-87

DTIC